Delphi object files decompiler
Delphi .NET object files decompiler

Andrey Mikhailov*
mikhailov@icc.ru

Alexey Hmelnov*
hmelnov@icc.ru

*Matrosov Institute for System Dynamics and Control Theory
Irkutsk

Ivannikov ISPRAS Open Conference 2017, Moscow
30 November, 2017
Decompilers

Tools

- **Partial decompilation**
  2. Delphi («DeDe (Delphi Decompiler)», «IDR», «EMS Source Rescuer»)

  *To parse the DCU, use the dcu32int tool*

- **Full decompilation**
  2. .NET («ILSpy», «NETReflector»)
Delphi object files

- «An **object file** is a file containing object code, meaning relocatable format machine code that is usually not directly executable. There are various formats for object files, and the same object code can be packaged in different object files. An object file may also work like a shared library. In addition to the object code itself, object files may contain metadata used for linking or debugging, including: information to resolve symbolic cross-references between different modules, relocation information, stack unwinding information, comments, program symbols, debugging or profiling information.»

- **DCU** = Delphi Compiled Unit. That is compiled *.*.pas file for x86

- **DCUIL** – that is compiled *.*.pas file for .NET

- **DCU32INT**₁ – Delphi unit parser

- **DCU ≥ OBJ ≥ EXE**

---

₁http://hmelnov.icc.ru/DCU/index.ru.html
The format of DCU files

Delphi object file unlike PE executable file has a more structured program representation, e.g. every procedure has its own memory block. It contains information about all the data types defined in the unit and it may include debugging information. DCUIL has small header file containing common information such as size, compile time, etc. The header is followed by tagged information. Tags are divided into the following groups:

- The list of the used units and dynamic libraries, including information about their definitions (of data types, procedures, etc) used in the unit.

- Information about the data types, procedures, variables, etc defined in the unit.

- The memory block, which contains the memory representation for procedures, constants, etc defined in the unit.

- The linking information for the memory block (where to place the addresses of some objects used when linking).

- Debugging information.
### Reached DCU decompilation level

<table>
<thead>
<tr>
<th>Platform</th>
<th>Source</th>
<th>Version</th>
<th>№</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win 32</td>
<td>x86</td>
<td>2.0 – 7.0, 2005 –</td>
<td>2 – 7,9 –</td>
<td>disassembler+dataflow</td>
</tr>
<tr>
<td>Win 64</td>
<td>x64</td>
<td>XE2 –</td>
<td>16 –</td>
<td>disassembler+dataflow</td>
</tr>
<tr>
<td>OS X,32</td>
<td>x86</td>
<td>XE2 –</td>
<td>16 –</td>
<td>disassembler</td>
</tr>
<tr>
<td>iOS,Simulator</td>
<td>x86</td>
<td>XE4 –</td>
<td>18 –</td>
<td>disassembler</td>
</tr>
<tr>
<td>iOS,Device</td>
<td>ARM 32</td>
<td>XE4 –</td>
<td>18 –</td>
<td>no</td>
</tr>
<tr>
<td>iOS,Device 64</td>
<td>ARM 64</td>
<td>XE8 –</td>
<td>22 –</td>
<td>no</td>
</tr>
<tr>
<td>Android</td>
<td>ARM 32</td>
<td>XE5 –</td>
<td>19 –</td>
<td>no</td>
</tr>
<tr>
<td>.NET</td>
<td>CIL</td>
<td>8.0 – 2006</td>
<td>8 – 10</td>
<td>decompiler</td>
</tr>
<tr>
<td>*</td>
<td>Inline</td>
<td>2005 –</td>
<td>9 –</td>
<td>decompiler</td>
</tr>
</tbody>
</table>

Note: The table shows the decompilation levels for different platforms and sources, along with version numbers and whether they are disassemblers, dataflow tools, or decoders.
## Decompilation phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Syntax analysis</td>
<td>Main task is determine the beginning and end of the «opcode»</td>
</tr>
<tr>
<td>2. Semantic analysis</td>
<td>We will assume that the object code is always semantically correct</td>
</tr>
<tr>
<td>3. Generic intermediate representation</td>
<td>For machine-independent Optimization</td>
</tr>
<tr>
<td>4. Control flow graph generation</td>
<td>Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in some optimization.</td>
</tr>
<tr>
<td>5. Data flow analysis</td>
<td>Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. The information gathered is often used by decompilers when optimizing a intermediate representation.</td>
</tr>
<tr>
<td>6. Control flow graph analysis</td>
<td>Recovering high-level control constructs is essential for decompilation in order to produce structured code that is suitable for human analysts and sourcebased program analysis techniques.</td>
</tr>
<tr>
<td>7. Code generation</td>
<td>Code generation can be considered as the final phase of decompilation. Source code generating from intermediate representation.</td>
</tr>
</tbody>
</table>
Delphi for .NET object code decompilation scheme

- Delphi object code
- DCU32INT tool
- DCU disassembler
- Control flow graph analysis
- Control flow graph generation
- Intermediate representation generation
- IR optimizations
- Delphi source code generation
- Delphi source code
Disassembler phase

The structure of the CIL command:

1. Can consist of one or two bytes
2. After the command, there may be metadata:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>0</td>
<td>The operand is empty</td>
</tr>
<tr>
<td>int8</td>
<td>1</td>
<td>A signed 8-bit integer</td>
</tr>
<tr>
<td>int32</td>
<td>4</td>
<td>A signed 32-bit integer</td>
</tr>
<tr>
<td>int64</td>
<td>8</td>
<td>A signed 64-bit integer</td>
</tr>
<tr>
<td>unsigned int8</td>
<td>1</td>
<td>Unsigned 8-bit integer</td>
</tr>
<tr>
<td>unsigned int16</td>
<td>2</td>
<td>Unsigned 16-bit integer</td>
</tr>
<tr>
<td>float32</td>
<td>4</td>
<td>32-bit floating-point number</td>
</tr>
<tr>
<td>float64</td>
<td>8</td>
<td>64-bit floating-point number</td>
</tr>
<tr>
<td>token</td>
<td>4</td>
<td>FixUp (address binding)</td>
</tr>
<tr>
<td>switch</td>
<td>variable</td>
<td>Array of jump addresses</td>
</tr>
</tbody>
</table>

Table — CIL operands
DCU for .NET disassembler implementation

CIL → TCILInstr.CILOpCode

TMethodBody – contain the sequence of TCILInstr

TCILExp − Abstract language representation

TCILOpCode = class
  protected
    Op1 : Byte;
    Op2 : Byte;
    Code : TCILCode;
    FlowControl : TFlowControl;
    OpCodeType : TOpCodeType;
   _operandType : TOperandType;
    StackBehaviorPop : TStackBehaviour;
    StackBehaviorPush : TStackBehaviour;
  public
    ...
end;

Mono — Mono is a software platform designed to allow developers to easily create cross platform applications part of the .NET Foundation

ILSpy — ILSpy is the open-source .NET assembly browser and decompiler
Disassembly algorithm

Data: Procedure memory block containing the CIL bytecode
Result: CIL sequence

while end of code do
    \( B \leftarrow \text{ReadByte}() \)
    if \( b \neq \$FE \) then
        \( \text{ByteCode} \leftarrow \text{OneByteOpCodeTbl}(B) \)
    end
    else
        \( B \leftarrow \text{ReadByte}() \)
        \( \text{ByteCode} \leftarrow \text{TwoByteOpCodeTbl}(B) \)
    end
    \( \text{ByteCode}.\text{ReadOperand}() \)
end
Intermediate representation 1. Pseudo register

ldlo c
ldlo c
mul
ldlo c
add
stlo c
x

MOV r1, &a
MOV r2, &b
MUL r1, r1, r2
MOV r2, &c
ADD r1, r1, r2
STORE r1, &x

st x
add
mul
ld c
ld b
ld a
Intermediate representation 2. Expressions

TCILInst.Expr ← TCILEexpr

Expressions

- TCILEexpr
  - TCILBinOp
  - TCILUnOp
  - TCILSemOp

All conditional or unconditional branches replace with

- TCILCondGoTo
- TCILUncondGoTo

Algorithm 1 Example. Callvirt method

1. An object reference obj is pushed onto the stack
2. Method arguments arg1 through argN are pushed onto the stack
3. Method arguments arg1 through argN and the object reference obj are popped from the stack; the method call is performed with these arguments and control is transferred to the method in obj referred to by the method metadata token. When complete, a return value is generated by the callee method and sent to the caller
4. The return value is pushed onto the stack
**Basic block** is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. The code in a basic block has:

1. **One entry point,** meaning no code within it is the destination of a jump instruction anywhere in the program.
2. **One exit point,** meaning only the last instruction can cause the program to begin executing code in a different basic block.

**Data:** A sequence of instructions

**Result:** A list of basic blocks with each three-address statement in exactly one block

1. The first instruction is a leader
2. The target of a conditional or an unconditional goto/jump instruction is a leader
3. The instruction that immediately follows a conditional goto/jump instruction is a leader
4. The first instruction of the exception block is the leader

Starting from a leader, the set of all following instructions until and not including the next leader is the basic block corresponding to the starting leader.

Creating edges

1. Calculate jump addresses
2. Create edges for
   - nodes with branch instructions
   - exceptions
Control flow analysis 1

Structuring Decompiled Graphs:  
- Edges are marked as direct, back, oblique
- Structuring Loops
- Structuring 2-way conditions (+ compound conditions)

---

2 Cifuentes C. Structuring decompiled graphs //International Conference on Compiler Construction. – Springer Berlin Heidelberg, 1996. – C. 91-105. MLA
Data: G, D, P
Result: an abstract node containing a hierarchy of folded subgraphs

\begin{verbatim}
foreach v ∈ D in a backward breadth-first order do
    foreach p ∈ Children(v) do
        if p pidom v then
            S ← Children(v) \ p
            if Classify_Region(S) ≠ undetermined then
                Apply_Template(S)
            end
        else
            Recognize_Undeterminanted_Region(S)
        end
    end
end
\end{verbatim}

Fig. – TT-region
Templates

Fig. – line

Fig. – loop

Fig. – self-loop

Fig. – if-then

Fig. – if-then-else

Fig. – switch
IR for regions

Regions

- TCILExpr
  - 1. TCILIfThenBlock
  - 2. TCILIfThenElseBlock
  - 3. TCILRepeatSt
  - 4. TCILWhileSt
  - 5. TCILCaseSt

TCILIfThenElseBlock = class (TCILExpr)
protected
  FTrue, FFalse: TCtrlFlowNode;
end
FCond: TCILCondition;
public
  constructor Create(ACond: TCILCondition; ATrue, AFalse: TCtrlFlowNode);
  destructor Destroy;
  function AsString(BrRq: boolean): String; override;
  procedure Show(BrRq: boolean); override;
  property Cond: TCILCondition read FCond;
Generation of expressions

Result := a + b + c + D;

...

ldarg.0 Pop: Pop0 Push: Push1 Type: InlineNone
ldarg.1 Pop: Pop0 Push: Push1 Type: InlineNone
add Pop: Pop1_push1 Push: Push1 Type: InlineNone
ldarg.2 Pop: Pop0 Push: Push1 Type: InlineNone
add Pop: Pop1_push1 Push: Push1 Type: InlineNone
ldloc.0 Pop: Pop0 Push: Push1 Type: InlineNone
add Pop: Pop1_push1 Push: Push1 Type: InlineNone
stloc.1 Pop: Pop1 Push: Push0 Type: InlineNone

CILCtx

- Locals – local variables
- Args – procedure arguments
- Stack – stack state
Features of data flow analysis

**Fig.** — The value is determined by one branch, used in different

**Fig.** — The value is defined in different branches, used in one

- **st** — push value on to the stack
- **ld** — load values from the stack
IR optimizations

- **Merging variables.** Elimination of intermediate calculations

- **Delete unused code.** Removing unreachable code, because it is impossible to determine the state of the stack

- **Copies propagation**
  1. Any instruction loading the address is copied to the "opcode" of its use
  2. Parameters propagation

- **Removing unused variables**

- **Simplify the instruction set for the jump instructions.** Jump commands are given to the general view (TCILCondGoTo, TCILUncondGoTo)

- **Combining complex logical expressions**
Test. Decompilation quality

$TS$ – set of test programs
$prog$ – source program
$KLOC(prog)$ – number of thousands of significant lines of the prog
$K$ – amount of penalties for the original program
$K'$ – amount of penalties for the decompiled program

\[ C_{decom} = \sum_{prog \in TS} \frac{\max(0, K' - K)}{KLOC(prog)} \]

(1)

<table>
<thead>
<tr>
<th>Name</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-recovery of variable name</td>
<td>1</td>
</tr>
<tr>
<td>goto operator</td>
<td>3</td>
</tr>
<tr>
<td>break operator</td>
<td>1</td>
</tr>
<tr>
<td>continue operator</td>
<td>1</td>
</tr>
<tr>
<td>non-recovery of for operator</td>
<td>1</td>
</tr>
</tbody>
</table>

Table – Penalties

*Troshina, «Issledovanie i razrabotka metodov dekompilyatsii programm», 2009 г.*
Test. Decompilation quality.

**Table – Decompilation quality**

<table>
<thead>
<tr>
<th>Name</th>
<th>DCUIL2PAS</th>
<th>ILSpy</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitWise</td>
<td>62.5</td>
<td>133.3</td>
</tr>
<tr>
<td>Compression</td>
<td>18.6</td>
<td>146</td>
</tr>
<tr>
<td>LZRW1KHCompressor</td>
<td>75</td>
<td>140</td>
</tr>
<tr>
<td>GetMatch</td>
<td>0</td>
<td>166.6</td>
</tr>
</tbody>
</table>

**Table – Performance**

<table>
<thead>
<tr>
<th>Name</th>
<th>files count</th>
<th>Size (mb)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delphi 8 VCL</td>
<td>325</td>
<td>39</td>
<td>396</td>
</tr>
</tbody>
</table>

**Table – Quality**

<table>
<thead>
<tr>
<th>Name</th>
<th>procedures count</th>
<th>without goto</th>
<th>with goto</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delphi 8 VCL</td>
<td>9003</td>
<td>8879</td>
<td>124</td>
<td>1.3</td>
</tr>
</tbody>
</table>
procedure TWinForm.btnCompress_Click(sender: System.Object; e: System.EventArgs);
var
  finfo : FileInfo; finput :FileStream; bwriter : BinaryWriter; ms : MemoryStream; fs : FileStream;
begin
  finfo := FileInfo.Create(textInput.Text);
  if (finfo.Exists) then begin
    finput := finfo.OpenRead();
    ms := MemoryStream.Create;
    bwriter := BinaryWriter.Create(ms);
    LZRWCompressFileToStream(finput, bwriter);
    if (bwriter <> nil) then begin
      fs := FileStream.Create(textOutput.Text, FileMode.Create);
      bwriter.BaseStream.Seek(0,SeekOrigin(0));
      (MemoryStream (bwriter.BaseStream)).WriteTo(fs);
      fs.Close();
      bwriter.Close();
    end;
    finput.Close();
  end;
end;

procedure TWinForm.btnCompress_Click (sender: Object; e: EventArgs);
var
  finfo: FileInfo; finput: FileStream; bwriter: BinaryWriter; ms: MemoryStream; fs: FileStream;
beg [Flags:3013,MaxStack:3,CodeSz:80,LocalVarSigTok:0]
  finfo := FileInfo.Create(Control.get_T ext(TWinForm.textInput));
  if (FileSystemInfo.get_Exists(finfo) <> 0) then begin
    finput := FileInfo.OpenRead(finfo);
    ms := MemoryStream.Create();
    bwriter := BinaryWriter.Create(ms);
    TWinForm.LZRWCompressFileToStream(Self, finput, bwriter);
    if (bwriter <> 0) then begin
      fs := FileStream.Create(Control.get_T ext(TWinForm.textOutput), 2);
      MemoryStream.WriteTo(MemoryStream(BinaryWriter.get_BaseStream(bwriter)), fs);
      FileStream.Close(fs);
      BinaryWriter.Close(bwriter);
    end;
    FileStream.Close(finput);
  end;
Delphi object files decompiler
Delphi .NET object files decompiler

Andrey Mikhailov*  Alexey Hmelnov*
mikhailov@icc.ru  hmelnov@icc.ru

*Matrosov Institute for System Dynamics and Control Theory
Irkutsk

Ivannikov ISPRAS Open Conference 2017, Moscow
30 November, 2017