2015 The 5th International Workshop on Computer Science and Engineering
doi: 10.18178/wcse.2015.04.033

Delphi .NET object file decompiler

Andrey A. Mikhaylov, Aleksey S. Burlakov, and Aleksey E. Hmelnov

Institute for System Dynamics and Control Theory of Siberian Branch of Russian Academy of Sciences (ISDCT SB RAS)

Keywords: Reverse engineering, object file, processor architecture, decompiler.

Abstract. We present DCUIL2PAS, a tool to decompile Delphi compiled units for .Net (*.dcuil files). Unlike DCU32INT, which disassembles Delphi compiled units and outputs machine code for subroutines, DCUIL2PAS is able to decompile Delphi compiled units for .Net, i.e. it produces more human understandable high level source code.

Introduction

The first decompiler was developed by Joel Donelly in 1960. His work resulted in the fundamental methods and means being applied in decompiler development nowadays. Although decompiling problem was set in the 1960th, it is still not completely solved. Disassembling and decompiling problem is algorithmic insolvable for many executable files, because it is impossible to separate data and code automatically. However, there are many works devoted to reverse engineering problems. The most significant ones are [1, 2], where the technics of decompilation of binary code were first proposed.

Delphi programmers widely use third-party components, which are distributed in the form of Delphi compiled units (DCU files), without source code. It allows component developers to distribute their components without disclosing their source. On the other hand, the third-party components make software less safe in terms of the information security. Moreover, the third-party components may have errors, which are difficult to fix, when the component source code is unavailable and its developer is unreachable for some reason. Another problem with compiled units is that they are Delphi version specific and can’t be used in the newer Delphi versions when their developer hasn’t published the DCU for that version.

The format of dcu (Delphi compiled unit) files is undocumented. In ISDCT SB RAS the DCU32INT tool [3] was developed. This tool allows to analyzes the dcu files of all the 32-bit Delphi versions starting from Delphi 2.0. It can parse dcu file and almost exactly restore the source code of the interface part of the corresponding unit, but the implementation part of the unit contains only disassembled machine instructions in subroutines` bodies, which is low level information. This information is hardly understandable and hardly allows to identify the subroutine algorithm structure.

There are many decompilers for .NET applications:

- .NET Reflector [4] used to be an open source project. At the moment it is the commercial project with closed source code. It supports C# and Visual Basic decompilation.
- ILSpy [5] is a free open source alternative to .NET reflector. This project have started when .NET Reflector got commercial.
- Dotnet IL Editor [6] is an open source project. The serious disadvantage of Dotnet IL Editor is that it does not support plugins.
- MonoDevelop [7] is an open source project, which unlike ILSpy does not support just in time navigation through the decompiled code.

The above decompilers generally take a .NET executable file as an input and produce C# code as an output. None of the decompilers support the format of DCU nor produces source code in Object Pascal.

The Format of DCUIL 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.

For DCUIL files the code blocks contains the CIL (Common Intermediate Language) instructions, which are used by .NET virtual machine. In comparison with a machine code of a real processor, the CIL byte code has the following advantages:

- The code and data are stored almost separately.
- The virtual machine is an abstract stack machine. The stack is strongly typed.
- The stack is usually used for intermediate results storing.
- The majority of CIL commands receive their own stack arguments, pop them from the stack, and push the result.
- The machine is object oriented: CIL commands support classes, methods, etc.

The DCUIL2PAS Decompiler Architecture

Figure 1 shows components of the decompiler as rectangles. The arrows represent dataflow and its direction. The DCUIL2PAS architecture is divided into the following blocks:

- The CILSeq module is an internal representation of an input program in the form of CIL instructions.
- The Control Flow Graph module builds the control flow graph.
- The Structural analysis module unit restores the high level language constructs and restructures accordingly the control flow graph.
- The Intermediate code generation module builds the intermediate representation of the program structure consisting of the structural basic blocks.
- The Code Optimization module manipulates the intermediate representation to improve the code generatinon quality. One of the important optimizations is restoration of complex conditions of conditional operators compiled in the short circuit boolean evaluation mode.
- The GUI module implements graphical user interface and supports syntax highlighting.

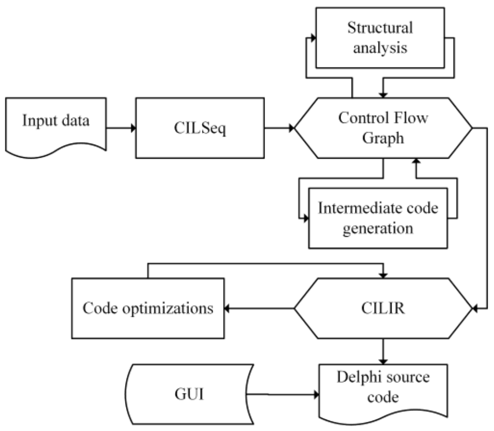

Fig. 1. DCUIL2PAS decompiler architecture.

The decompiler takes DCUIL file as an input parameter. It locates the subroutines` code blocks using the DCU32INT functions. To disassemble the CIL byte code the Mono [4] disassembler library is used. Then the control flow graph is built by detecting the basic blocks [8] and their relations. An intermediate representation is built for every basic block. Each instruction is associated with the expression entirely describing its semantics. All the computational instructions take their arguments from the stack and place their results on the top of the stack. So the expressions for neighbour instructions performing computations on the stack can be combined into more complex expressions. Then the structural analysis of the graph of basic blocks is carried out by the methods of dominance tree [9]. An intermediate representation obtained is then optimized to improve the quality of the resulting code. As an output, the decompiler produces Delphi code which is semantically equivalent to the source code.

Intermediate Representation

The CIL intermediate representation used in DCUIL2PAS is represented by a hierarchy of classes and it is built for every basic block of the control flow graph. First the initial data state for each block is determined. It consists of the states of function argument`s, local variables, and the stack. Then all the CIL instructions in the linear code fragment of the block are associated with the expressions corresponding to their opcode semantics.

Since a variable in a fragment of linear code may have several entries in expressions, every variable gets its own reference counter when a new expression is generated. In the beginning, all the reference counters are set to 1. Each entry of the variable increments the counter by 1. When the variable is redefined it gets a reference to a new object and the counter resets to 1. After that, in order to obtain the results of expressions, the references to the existing objects are used.

Figure 2 depicts the class hierarchy implementing the intermediate representation. TCILExpr base class implements counting reference and defines the common methods:

- Eval – evaluates the expression.
- Eq(E: TCILExpr) – returns true if the parameter of the method is equals to current expression.
- AsString(BrRq: boolean) – returns the text representation of the expression.
- Show – prints the text of the expression.

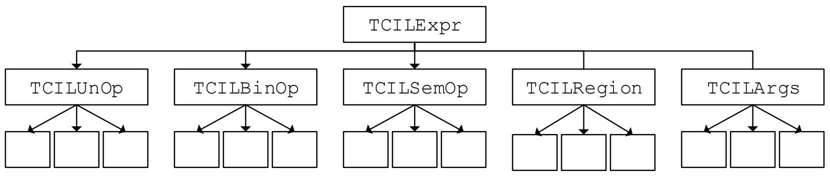

Fig. 2. Intermediate representation classes' hierarchy.

TCILUnOp class successors specify all the single-argument opcodes. Similarly, TCILBinOP successors specify the semantic of all the binary operations. The expressions for arguments and local variables inherit from TCILArgs class. TCILSemOp is used to specify all the other instructions.

The regions detected by the control flow analysis are represented by the successors of the `TCILExpr` base class. These classes are: `TCILIfThenElse, TCILWhile, TCILRepeat, TCILCaseSt, TCILBasicBlack.`

The compiler can use two different methods for compilation of logical expressions:

- Complete boolean evaluation. The logical operations are computed by the corresponding CIL instructions on the stack, their arguments are always evaluated, hence the term "complete". The logical result is popped later from the stack as an argument of a conditional branching opcode.
- Short circuit Boolean evaluation. The code generation for conditional expressions is based upon the following rules:

- `A` **and** `B => ` **if** `A` **then** `B` **else** `False`,
- `A` **or** `B => ` **if** `A` **then** `True` **else** `B`.

As a result the computation of some expression parts may be skipped, hence the term “short circuit”.

For the complete boolean evaluation the code generation is straightforward, because the expression corresponding to the logical value popped from the stack represents the original logical condition. For the short circuit Boolean evaluation, it is required to restore the original logical expression from the graph of branching instructions and their conditions using the predetermined set of rules, e.g. `If Then Else(Cond, If Then Else(Cond1, lblTrue1, lblFalse), lblFalse)` expression would be reduced to `If Then Else(Cond and Cond1, True1, False)`. When a False branch contains `If Then Else(Cond, If Then Else(True, (Cond1, True, False)), False)` conditional expression, then the expression will be reduced to `If Then Else(Cond or Cond1, True, False)`. The cycles are combined similarly. `While St(Cond, While St (Cond1,Trg))` will be reduced to `While St(Cond and Cond1, Trg)`.

For example, intermediate representation for the code:

```
function IfSt1 (a: Integer; b: Integer; c: Integer): Integer;
var
 Result: Integer;
begin
 if (a > b) then
   if (a > c) then
     if (b > c) then
       Result := 1
     else
       Result := 0
   else
     Result := 0
 else
   Result := 0;
end;
```

will look as follows:

```
IfThenElse(Cond, IfThenElse(Cond1, IfThenElse(Cond2, True, False), False), False)
```

If we optimize combinations of conditions then the following code will be generated:

```
function  IfSt1 (a: Integer; b: Integer; c: Integer): Integer;
var
 Result: Integer;
begin
 if (a > b) and (a > c) and (b > c) then
   Result := 1;
 else
   Result := 0;
end;
```

Example of Decompilation

The relevance of the work is demonstrated below. The following listing presents the result of analyze made by DCU32INT tool of the Fact.dcuil sample (containing a factorial computation function).

```
function  Fact (n: Borland.Delphi.System.Integer):
 Borland.Delphi.System.Integer;
var
 Result: Borland.Delphi.System.Integer;
 i: Borland.Delphi.System.Integer;
   : Borland.Delphi.System.Integer;
begin
 00: .      |02                     | ldarg.0
 01: .      |17                     | ldc.i4.1
 02: /.     |2F 04                | bge.s  $8
 04: .      |16                   | ldc.i4.0
 05: .      |0C                   | stloc.2
 06: +.     |2B 1A                | br.s  $22
 08: .      |17                   | ldc.i4.1
 09: .      |0C                   | stloc.2
 0A: .      |17                   | ldc.i4.1
 0B: .      |02                   | ldarg.0
 0C: .      |0A                   | stloc.0
 0D: .      |0B                   | stloc.1
 0E: .      |06                   | ldloc.0
 0F: .      |07                   | ldloc.1
 10: 2.     |32 10                | blt.s  $22
 12: .      |06                   | ldloc.0
 13: .      |17                   | ldc.i4.1
 14: X      |58                   | add
 15: .      |0A                   | stloc.0
 16: .      |08                   | ldloc.2
 17: .      |07                   | ldloc.1
 18: Z      |5A                   | mul
 19: .      |0C                   | stloc.2
 1A: .      |07                   | ldloc.1
 1B: .      |17                   | ldc.i4.1
 1C: X      |58                   | add
 1D: .      |0B                   | stloc.1
 1E: .      |07                   | ldloc.1
 1F: .      |06                   | ldloc.0
 20: 3ф     |33 F4                | bne.un.s  $16
 22: .      |08                   | ldloc.2
 23: *      |2A                   | ret
end ;
```

This output source code is not enough abstract and it is very complex to analyze. In order to restore the high level source code it is necessary to know the stack condition, variables' and function arguments' values at each moment of time. Moreover we need to know the semantics of all the CIL opcodes.

The listing below shows the result of decompilation of the function by DCUIL2PAS. Unlike the previous result, this one does not contain low level instructions, hence it is more abstract and human understandable.

```
function  Fact (n: Integer): Integer;
var
  LocVar: integer;
  i: integer;
begin
   if (n < 1) then
     Result := 0
   else begin
     Result := 1;
     LocVar := n;
     i := 1;
     if (LocVar >= i) then begin
       LocVar := LocVar + 1;
       while (i <> LocVar) do begin
         Result := Result * i;
         i := i + 1;
       end;
     end;
   end;
end;
```

Conclusions

The intermediate representation which generates the code semantically equal to Object Pascal code was proposed and implemented. The Dcuil object files were used as an example of the object files, which contain additional information about the code in comparison with executables and also use a rather simple byte code. The high level of abstraction of the CIL byte code with the information about the data types and variable names contained in the dcuil files makes it possible to generate the code of high quality. The primary goal of the future work is to expand the results obtained to the DCU files with the more complex byte code such as that of Intel 80x86 instructions.

References

[1] http://boomerang.sourceforge.net // A general, open source, retargetable decompiler of machine code programs.

[2] http://www.itee.uq.edu.au/cristina/d-cc.html#thesis // Cifuentes C. Reverse compilation techniques. — 1994.

[3] http://hmelnov.icc.ru/DCU/index.ru.html // DCU32INT - Delphi Compiled Units Parser.

[4] Reflector .NET. – URL: http:// www.red-gate.com/products/dotnet - development/reflector/

[5] ILSpy. – URL: http ://ilspy.net/

[6] Editor Dotnet IL. – URL: http://sourceforge.net /projects/dile

[7] MonoDevelop. — URL: http://monodevelop. com. http://www.mono-project.com/Main_Page // cross platform, open source .NET development framework.

[8] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986. ISBN 0-201-10088-6

[9] C. Cifuentes. A structuring algorithm for decompilation. In Proceedings of the XIX Conferencia Latinoamericana de Informatica, pages 267{276, Buenos Aires, Argentina, 2-6 August 1993. Centro Latinoamericano de Estudios en Informatica.