DECOMPILATION IN .NET FRAMEWORK

Pavel Fiala, Miroslav Virius
FJFI ČVUT v Praze, fialapa5@fjfi.cvut.cz

ABSTRACT:
This article explains the process of decompilation in .NET Framework and introduces the design of abstract syntax tree representation in XML.

KEYWORDS:
MSIL, abstract syntax tree, assembly decompilation, XML representation

INTRODUCTION: EXISTING SOLUTIONS AND MOTIVATION

There are decompilation programs for the .NET Framework. One the famous decompilation tool for the .NET is the Lutz Roeder's .NET Reflektor [5], now distributed by the RedGate Company; this tool is paid since 2011. The Dis# distributed by the NETdecompiler.com Company [6] is paid, too. The Salamander .NET Decompiler distributed by Remotesoft [7] is paid as well. Algorithms used by these tools were not published.
Our aim is to develop a flexible tool for the decompilation based on XSL transformation of the abstract syntax tree (AST) and to make the algorithms available to public. The program shall be able to read the user-defined transformation description for any programming language and to produce the source code according to this transformation.
In the first part of this article, we briefly explain the algorithm for the creation of the AST representing the decompiled assembly [1-4]. The next part presents our design of the XML representation of the AST.

ASSEBMLY TRANSFORMATION TO THE ABSTRACT SYNTAX TREE

The basic idea of the decompilation of the .NET assembly is to create the abstract syntax tree according to the information stored in the assembly and to transform this tree. Tools from Microsoft .NET Framework can be used to simplify the decompilation process. The classes from `System.Reflection` and `System.Reflection.Emit` namespaces are very useful for extracting the assembly content. There are two main ways of the decompilation. The whole assembly can be processed and stored in the AST or only the sketch of the assembly (assembly attributes, modules, types, type's fields and methods) is stored and the rest of the assembly is processed according to the incoming requests and added to the stored tree. We will use the second way.

At the beginning of the decompilation, the assembly is loaded by the method provided in the `System.Reflection.Assembly` class and the root element of the whole XML document is created. Now, we can obtain both information about assembly declaration, the module list which the assembly consists of, referenced external assemblies and used resources. First, all of assembly attributes are added to the XML document, and then resources and references are added to the XML document. Last, the module list is obtained via `GetModules()` method.

Each module is processed in the following way. First, the list of global methods is obtained from the module. Next, the list is parsed into the dedicated XML elements. The whole content of the module has to be member of some namespace. If the method's namespace doesn't exist in the AST, it's created and added to the module's XML representation. Then, method element is added to the namespace element. After the global methods, the list of types is got from the module. The type element is created now.

Now, the base type and the list of implemented interfaces are obtained. The base type can be only one so it's added as the child element. We have to reduce the interface list before adding the interfaces to the XML because we want to show only interfaces implemented in this type and not the ones implemented in the base type. This reduced list is inserted to the type XML element then. Moreover, the type can have generic parameters thus they are stored in the dedicated element. Then the most important parts of the type are still waiting to be processed – fields, properties, methods and events. They are got via `GetMembers()` method and they are parsed to the XML elements according to their type.

For the field, the full element is created. For the property, the element with the full information about the name, the attributes and the parameters is created. Then only the sketch of the property member is created. That means that only the name and the type are stored. Similar situation is for the method – only the sketch is created. The method element has almost its full content except the method body. That element is constructed after any incoming request. Event for the type has the full XML element. Now, the first stage is finished and the basic AST is done.

Next step is the evaluating of the method nodes according to the incoming request. If the request is for the non-evaluated method, it is processed. The method node contains everything important except the method body. Since MSIL is similar to the assembler, function invoking with the input parameters and returning value is done by stack (and many other operation use stack too). Hence the stack operations have to be simulated as they are done in the .NET virtual machine. Method body has a special class `System.Reflection.MethodBody`. Using this class, it is possible to get three important things: maximum stack size, local declarations (only the list with variable types without names) and an array of bytes representing the code. Now, we are ready to pass through the byte array. We are at the position 0 at the beginning. The first byte is read and if the value is 0xFE, the second byte is read immediately. According to the byte value, the operation and its operand are recognized. If the operand is given in the byte array, it is read and the position is increased by the specified value for this operation and the stack operation is simulated. We have prepared the state machine to determine constructions according to the operation sequences. When the stack work is done, the state is refreshed according to the operation type, and when the final state is reached, appropriate XML element is created and added to the method body element. The whole method body is processed this way.

ABSTRACT SYNTAX TREE REPRESENTATION IN XML

We designed the abstract syntax tree representation in the XML according to the analysis of the permitted constructions in MSIL.

The first and basic element is an `assembly` element.

```
<assembly name="">
  <assemblyDeclaration name="" value="" />
  <module />
  <resource name=”” />
  <reference name=”” />
</assembly>
```

This element has only the `name` attribute, which contains the name of the analysed assembly. The `assemblyDeclaration` child elements represent the information about the assembly declarations (i.e. assembly version, copyright etc.). Next, `module` elements contain the information about the modules of the analysed assembly. Last two elements, `resource` and `reference`, are optional and they hold the resources used and referenced external assemblies.

```
<module name="">
  <namespace name="">
    <method />
    <type />
  </namespace>
</module>
```

Structure of the `module` element is simple – each module has name (attribute `name`) and the whole module content has to be member of some namespace. For the namespace, there is element `namespace` with only one important attribute – its name. In this element, there are `method` child elements for any method declared outside of any type and `type` elements for types declared in the module. Both type and method element can occur more times.

```
<type name="">
  <extends name="" />
  <implements name="" />
  <genericParams />
  <field />
  <property />
  <method />
  <event />
</type>
```

For the type, it's important to know its name (`name` attribute in the root `type` element). First two child elements, `extends` and `implements`, contain the information about the base type of this type and about interfaces which are implemented in this type; `implements` attribute can occur more times. Next, it's important to have generic parameters (if there is any) in `genericParams` element, field, properties and methods of the type (in the `field`, `property`, `method` elements).

```
<genericParams>
  <genericParam type=”” name=””>
    <genericParamAttribute name="" />
  </genericParam>
</genericParams>
```

The element `genericParams` contains the list of type's genetic parameters; each item is in the `genericParam` element. For generic parameter, we have to keep its attribute, type and name.

```
<field type="" name="" array=”” >
  <genericParams />
  <fieldAttribute name="" />
</field>
```

Type can consist of the definition of the following items: fields, properties and methods. For the field or variable, there is the `field` element. There is important element for field attributes, generic parameters element and attribute informing whether this field is an array or not.

```
<property name="" type="" >
  <propertyAttribute name="" />
  <propertyArgument>
    <parameter />
  </propertyArgument>
  <propertyMember type="" name="" >
    <parameter />
    <codeBlock />
  </propertyMember>
</property>
```

Property is a special construction, basically it is a member variable but there are methods defined for setting and getting its value and other actions. It has the `name` and `type` attributes in the root element. As it is a combination of the variable and methods, this element has child elements for its attributes, arguments for the actions and the actions itself.

```
<method name="" returnType=”” >
  <methodAttribute name="" />
  <genericParams />
  <paramList>
    <parameter />
  </paramList>
  <methodBody>
    <localDeclaration />
    <codeBlock />
  </methodBody>
</method>
```

Methods are very important part of the assembly and we have the `method` elements for this construction. In the root element, there are the `name` attribute for the method name and `returnType` to specify the return value type. Optional child elements for method are `methodAttribute` elements. Type can have generic parameters. So can the method. Moreover, method can have input arguments and they are stored in the `paramList` element with the `parameter` as the child elements. Next, there is method body with declaration of local variables and block of the method code.

```
<delegate name="" returnType=”” >
  <delegateAttribute name="" />
  <paramList />
</delegate>
```

The special construction is delegate. It is like “pointer to function”. For this, it is important to know the name, return type, attributes and input arguments.

```
<event name="" type=”” >
  <eventAttribute type="" />
</event>
```

Except the methods, any type can contain the declaration of the event and there is the `event` element for its structure. It contains the `event` attributes and attributes of its name and type.

```
<parameter type="" name="">
  <paramAttribute type="" />
</parameter>
```

The `parameter` element for parameters is very simple. It consists of the name and type of the parameter and of the list of `parameter` attributes.

```
<localDeclaration>
  <field />
</localDeclaration>
```

For the local declaration, there is `localDeclaration` element which contains the list of the fields.

```
<codeBlock>
</codeBlock>
```

Very large part of the syntax tree in XML is the block of the code. We prepared very general element `codeBlock` which holds a group of statements.

```
<unaryStatement>
  <operand />
  <operation />
</unaryStatement>
<binaryStatement>
  <operandLeft />
  <operandRight />
  <operation />
</binaryStatement>
```

Description of statements in the code block starts with the unary or binary statement elements. The unary statement is composed of an operand and an operation, the binary statement has a second operand moreover.

```
<assignStatement>
  <leftSide />
  <rightSide />
</assignStatement>
```

The syntax of assignment statement reflects the `assignStatement` element which has two child elements – for left and right hand side.

```
<arrayElement name="">
  <index />
</arrayElement>
```

Simple variable identifiers can be presented in these elements. Array can be present in these statements. We have the `arrayElement` element with the `name` attribute for the array identifier and the `index` child element for the position in the array. This can be either the number or the returned function value.

Now, we will introduce other possible constructions – i.e. simple items as function invocation or items from high-level programming languages (if statements, loops etc.).

```
<ifStatement>
  <condition />
  <ifTrueBlock />
  <ifFalseBlock />
</ifStatement>
```

The if statement is composed of a condition and two code blocks – the `ifTrueBlock` when the condition is true and the `ifFalseBlock` otherwise. These two child elements are of the `codeBlock` type with different names only.

```
<switchStatement>
  <testItem />
  <case>
    <value />
    <codeBlock />
  </case>
</switchStatement>
```

The switch statement is a little bit similar to the if statement. The `switchStatement` element contains the `testItem` element which specifies the test item for the condition and the `case` child elements which have the `value` element for the condition and codeBlock in the case the condition is true.

Many high-level languages or assembly languages support loops, at least the while loop and the for loop. Moreover, C# supports modified for loop named foreach.

```
<whileStatement>
  <condition />
  <codeBlock />
</whileStatement>
```

The while loop and its element `whileStatement` contain `condition`. If it is fulfilled, the `codeBlock` is executed.

```
<forStatement>
  <localDeclaration />
  <condition />
  <incrementation />
  <codeBlock />
</forStatement>
```

Structure of the `forStatement` element is similar to the for statement syntax in many programming languages. It can contain the local declaration, condition for the execution of the `codeBlock` and the incrementation section. All the three elements are optional, `codeBlock` is mandatory.

```
<foreachStatement>
  <localDeclaration />
  <array />
  <codeBlock />
</foreachStatement>
```

The foreach statement is designed for simple sequence collection iteration. The iterated collection is stored in the `array` element; the variable for the collection item is described in the `localDeclaration` element. The same code is executed for each collection item. This code is described in the `codeBlock` element.

```
<condition type="">
  <leftSide />
  <rightSide />
</condition>
```

In the loops or in the if statement, there is one special construction – a condition. We have the `condition` element for this situation. It has the `type` attribute and two child elements for the left and the right hand side of the statement.

```
<goToStatement target="">
```

The goto statement to jump to the different line of the code is permitted in many higher level programming languages. The element for this command is very simple with one `target` attribute.

```
<castStatement type="" >
  <statement />
</castStatement>
```

The element `castStatement` reflects type casting with `type` attribute which determines the target type. The child element statement determines the value which is supposed to be casted to the target type.

```
<tryStatement>
  <codeBlock />
  <catchBlock clauseReference="" >
    <codeBlock />
  </catchBlock>
  <faultBlock clauseReference="" >
    <codeBlock />
  </faultBlock>
  <filterBlock>
    <codeBlock />
  </filterBlock>
  <finallyBlock>
    <codeBlock />
  </finallyBlock>
</tryStatement>
```

The syntax of the block for catching thrown exceptions is the base of the `tryStatement` element which is composed of blocks of code. First block with `codeBlock` element is designed for the possibly dangerous code where the exception can be thrown. There has to follow at least one `catchBlock` element to handle thrown exception or the `finallyBlock` element after this element. The `catchBlock` element has one `clauseReference` attribute which provides the type of the exception to be handled. In some programming languages, there are two special constructions that are executed after catch blocks – filter and fault block. All languages for .NET support finally block so there is an element for this construction.

```
<methodCall name="">
  <paramList />
</methodCall>
```

Very important thing is method call. The `methodCall` element can occur separately or as a part of other elements. This element has the `name` attribute and a child element for method parameters.

CONCLUSION: RESULTS AND FUTURE WORK

We have presented our decompilation algorithm and the list of the XML elements reflecting the assembly structure. This is the crucial part of our design of the developed tool.

We have implemented the presented algorithm using our AST representation in XML and we are going to implement XSL transformations now. Our goal is to create template of XSL styles for the transformations of the XML representation of the AST to the source code of the selected programming language.

Our first step will be to create these styles for the MSIL and C# languages. Next, we suppose to implement the decompilation transform for the Visual Basic and C++/CLI.

ACKNOWLEDGEMENT
The work on this article was supported by the SGS11/167 grant of the Ministry of the Education, Youths and Sports of the Czech Republic.

REFERENCES

1. *NET Framework Developer's Guide – Inside the .NET Framework.* URL: <http://msdn.microsoft.com/en-us/library/a4t23ktk(VS.71).aspx> [cit. 2011-03-11]
2. Fiala P: *Analýza sestavení .NET*. Praha: ČVUT 2010.
3. Gough J. *Compiling for the .NET Common Language Runtime*. Prentice Hall 2001. ISBN 0-13-062296-6
4. Fiala P., Virius, M. *Strom abstraktní syntaxe a dekompilace MSIL*. In: Tvorba softwaru 2010. Ostrava: VŠB – TU Ostrava 2010. p. 47.
5. *.NET Reflector*. URL: < http://reflector.red-gate.com/> [cit. 2011-04-06]
6. *Dis#*. URL: <http://netdecompiler.com/> [cit. 2011-03-21]
7. *Salamander .NET decompiler*. URL: <http://www.remotesoft.com/salamander/index.html>[cit. 2011-03-21]