# XML Externalization Built into Compiler Front-Ends Using a Parser Generator

Kazuaki Maeda *

*Abstract*—**This paper describes XML externalization built into compiler front-ends and its application to quick reverse engineering tool development. A parser generator MoJay was developed to build XML externalization functionality into compiler front-ends. After replacing the original parser generator with MoJay, generating a parser using it, and modifying a few lines of source code in the compiler, we were able to obtain a special compiler that externalizes three types of information in the form of XML documents, namely, lexical information, syntactic information, and parse tree. The syntactic information was applied to develop a reverse engineering tool for C#. The tool shows a performance penalty from the viewpoint of the generated XML document size. However, even with a storage penalty, the quick development is a far superior option.**

*Index Terms* — **parser generator, reverse engineering tool, XML, C#**

## 1 Introduction

The growth in computing power and the proliferation of the Internet have made XML a very popular tool for the representation and exchange of data. Today, the use of XML has spread across many fields of applications. For example, it is used for setting application configurations, storing data in databases, retrieving data from databases, exchanging data over the Internet, invoking remote methods, et al.

XML is a markup language derived from the standard generalized markup language (SGML), and it is designed to be a text-based, human-readable, and self-describing language. It is independent of all platforms; hence, it can be used across different computers, different operating systems, and different programming languages.

The specification of XML does not restrict any specific libraries to process XML documents. If the libraries conform to XML standards, any tools based on the libraries can read, analyze, and write XML documents. In order to process XML documents, many libraries have already been implemented for a majority of the programming languages.

The orientation of XML documents is generally either document-centric or data-centric[1, 2]. The aim of the document-centric XML documents is visual consumption, and hence, they have less structured characteristics. Books, articles, and emails are the typical examples of document-centric XML documents. XHTML is a language to describe web pages as document-centric XML documents.

In contrast, data-centric XML documents typically include very granular collections of data, and they are applied to computer processing and database storage. For example, bibliography data and order forms are typical examples of data-centric XML documents. The data exchanged with web services is mostly data-centric. Hereafter, a data-centric XML document will be referred to as "XML data" in this paper.

The compiler is a traditional basic software that is indispensable for developing software. The main purpose of a compiler is only the generation of efficient object code. However, there are rare cases where a compiler is used for different purposes from the code generation. The compiler includes excellent algorithms and valuable information based on the results of years of research. This paper describes XML externalization built into compiler front-ends by using a parser generator MoJay and its application to the quick development of a reverse engineering tool.

In section 2, the modification of a free and open source compiler and a reverse engineering tool will be explained. In section 3, the XML representation of information in compiler front-ends will be explained. In section 4, the quick development of a reverse engineering tool using XML will be explained. The final section is the summary of this paper.

*This research was partly supported by a grant of the Open Research Center Promotion Project from Ministry of Education, Culture, Sports, Science and Technology in Japan. The contact information of the author is Department of Business Administration and Information Science, Chubu University, 1200 Matsumoto, Kasugai, Aichi 487-8501, Japan, Tel: +81-568-51-1111, Fax: +81-568-52-1505, Email: kaz@acm.org.

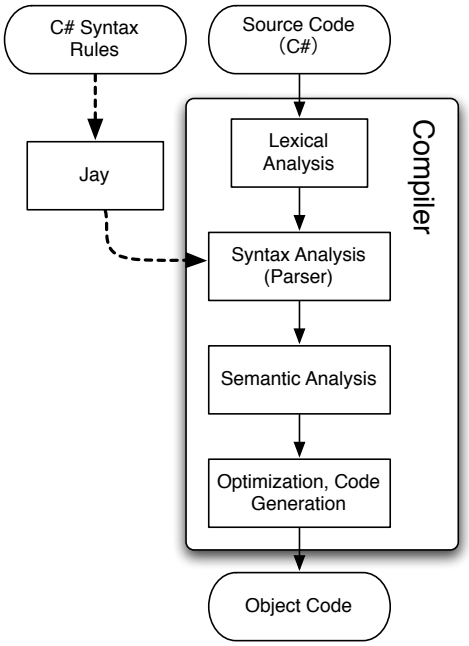


Figure 1: Conceptual structure of the typical compiler

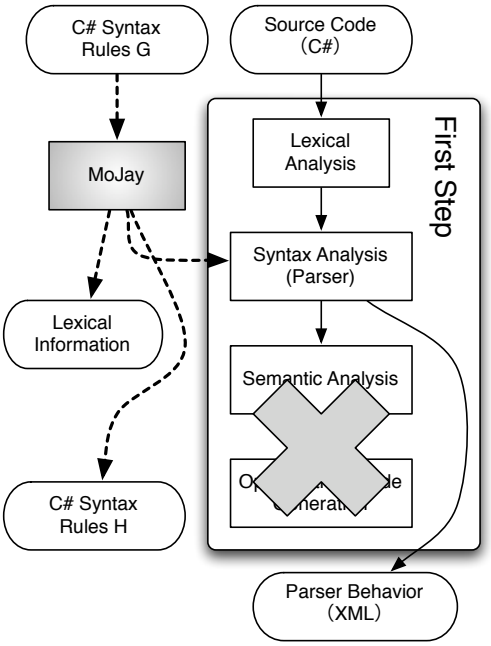


Figure 2: Modified version of the Mono C# compiler

## 2 Background and Motivation

### 2.1 Modified Mono C# Compiler

Mono[3] is an open source implementation of Microsoft .NET development environment and tools. It provides a C# compiler called "gmcs," which complies with the C# 2.0 language specification[4]. In order to implement the compiler, there are some classes for lexical analysis, syntax analysis, semantic analysis, optimization, and code generation, as shown in Figure 1. Jay is used to develop the syntax analysis program (i.e., parser) in gmcs. Jay is a parser generator that accepts Yacc-compatible syntax rules[5]. It reads syntax rules for C# programming language and generates a parser written in C# to analyze C# source code.

In order to build an XML externalization functionality into the compiler, Jay is replaced with a parser generator MoJay, which has been developed by the author. After replacing Jay with MoJay, generating a parser by MoJay, and modifying a few lines of source code in the compiler, we obtain a special version of gmcs, as shown in Figure 2. In this paper, this special version is referred to as "mocs." It externalizes three types of XML data after executing the compiler front-ends. Additional information is described in section 4.1.

### 2.2 Reverse Engineering Tools and Compiler Front-ends

In many cases the system design documents are not updated after the source code is modified. Reverse engineering tools have been developed as a solution for finding discrepancies between the source code and design documents.

A basic function of the reverse engineering tools is to generate class diagrams from the source code. For example, a reverse engineering tool for C#, which the author developed in C#, can generate class diagrams from the source code, as shown in Figure 3. The class diagram in the figure shows a part of the class hierarchies for the Mono C# compiler gmcs [1].

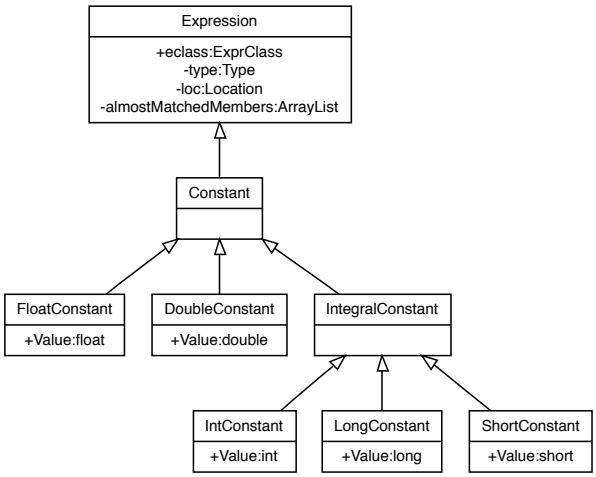


Figure 3: A generated class diagram for the Mono C# compiler

[1] Methods in the classes are intentionally eliminated.

```
using System;
namespace Com.Xyz
{
    public class Hello
    {
    }
}
```

Figure 4: A sample C# source code

In order to develop the reverse engineering tools, capability similar to a compiler front-end must be developed to analyze the source code. A typical compiler front-end reads the source code and executes lexical analysis, syntax analysis, and semantic analysis, as shown in Figure 1. However, the development of the compiler front-end capability involves some complicated tasks because the specifications of modern programming languages are becoming more complex with each passing year. Much workload is required develop it according to the language specification.

This paper describes XML externalization built into compiler front-ends and its application to quick reverse engineering tool development.

## 3 Externalizing Information in the Form of XML Data

There are three types of information in the compiler in the form of XML data, which is externalized by mocs, lexical information, syntactic information, and parse tree. These XML data are useful to develop reverse engineering tools.

### 3.1 XML Data of Lexical Analysis

A sample C# source code is shown in Figure 4. It is intentionally simplified in order to explain externalizing information in the form of XML data in this section. After the modified C# compiler mocs reads the C# source code, it externalizes lexical XML data, as shown in Figure 5. The externalized XML data includes the types of lexical items, string images, line numbers, and column numbers. By using only the XML data, it is possible to reconstruct the C# source code. It is also possible to generate the HTML document for the C# source code in color, e.g., blue italic keywords and green comments when the C# source code is reconstructed.

### 3.2 XML Data of Syntax Analysis

The parser generated by MoJay is a bottom-up one based on LALR, which is the same as Yacc[5]. It executes some actions such as reading a token, shifting a token, and reducing a rule during the analysis of the source code. When the modified C# compiler mocs analyzes

```
<lex st="0" tk="USING" va="using" li="1" co="1" />
<lex st="3" tk="IDENTIFIER" va="System" li="1" co="7" />
<lex st="30" tk="SEMICOLON" va=";" li="1" co="13" />
<lex st="6" tk="NAMESPACE" va="namespace" li="2" co="1" />
<lex st="50" tk="IDENTIFIER" va="Com" li="2" co="11" />
<lex st="72" tk="DOT" va="." li="2" co="14" />
<lex st="70" tk="IDENTIFIER" va="Xyz" li="2" co="15" />
<lex st="80" tk="OPEN_BRACE" va="{" li="3" co="1" />
<lex st="468" tk="PUBLIC" va="public" li="4" co="5" />
<lex st="63" tk="CLASS" va="class" li="4" co="12" />
<lex st="206" tk="IDENTIFIER" va="Hello" li="4" co="18" />
<lex st="495" tk="OPEN_BRACE" va="{" li="5" co="5" />
<lex st="729" tk="CLOSE_BRACE" va="}" li="6" co="5" />
<lex st="785" tk="CLOSE_BRACE" va="}" li="7" co="1" />
```

Figure 5: An example of lexical XML data

the source code, it externalizes XML data, as shown in Figure 6. The XML data represents the bottom-up parser actions, which is referred to as "parser behavior" in this paper. The parser behavior records a sequence of parser actions in XML during the syntax analysis. Table 1 shows the names and meanings of the elements, and Table 2 shows the names and meanings of their attributes.

```
<parse name="hello.cs">
<lex st="0" tk="USING" va="using" li="1" co="1" />
<shi fr="0" to="3" />
<lex st="3" tk="IDENTIFIER" va="System" li="1" co="7" />
<shi fr="3" to="30" />
<lex st="30" tk="SEMICOLON" va=";" li="1" co="13" />
<red st="30" ru="319" />
<red st="33" ru="316" />
<red st="31" ru="28" />
<shi fr="32" to="71" />
<red st="71" ru="21" />
<red st="14" ru="18" />
<red st="11" ru="10" />
<red st="9" ru="7" />
<lex st="6" tk="NAMESPACE" va="namespace" li="2" co="1" />
<red st="6" ru="53" />
<shi fr="16" to="50" />
<lex st="50" tk="IDENTIFIER" va="Com" li="2" co="11" />
<shi fr="50" to="72" />
<lex st="72" tk="DOT" va="." li="2" co="14" />
<red st="72" ru="319" />
<red st="33" ru="316" />
<shi fr="80" to="70" />
<lex st="70" tk="IDENTIFIER" va="Xyz" li="2" co="15" />
<shi fr="70" to="116" />
<red st="116" ru="318" />
<lex st="80" tk="OPEN_BRACE" va="{" li="3" co="1" />
<red st="80" ru="22" />
<shi fr="196" to="294" />
<red st="294" ru="29" />
<lex st="468" tk="PUBLIC" va="public" li="4" co="5" />
```

Figure 6: An example of parser behavior in XML

### 3.3 XML Data Representing a Parse Tree

When a typical compiler completes the syntax analysis, the parser builds up an abstract syntax tree, which correctly represents the hierarchical syntactic structure of the source code. Compiler developers need to embed

Table 1: Elements in the parser behavior

| Name | Meaning of the element |
|---|---|
| lex | reading a token |
| shi | shift action |
| red | reduce action |
| xdc | XML documentation comment |
| acc | acceptance |

Table 2: Attributes in the parser behavior

| Name | Meaning of the attribute |
|---|---|
| st | state number |
| fr | source state number for shift action |
| to | destination state number for shift action |
| tk | kind of a token |
| va | string image of a token |
| li | line number |
| co | column number |
| ru | syntax rule number |

functions in the parser to build an abstract syntax tree.

A parse tree is a representation of a derivation that filters the order in which productions are applied to replace non-terminals[6]. If the syntax rules of a programming language are defined, the parse tree can be automatically constructed by analyzing the source code. However, the parse tree is not used as a result of syntax analysis; instead, it can be only used to explain syntax analysis in textbooks on compiler construction, or it is sometimes used to debug the parser. The modified C# compiler mocs writes out the parse tree in the form of XML data, as shown in Figure 7.

```
<compilation_unit>
  <outer_declarations>
    <outer_declarations>
      <outer_declaration>
        <using_directive>
          <using_namespace_directive>
            <USING va="using"/>
            <namespace_name>
              <namespace_or_type_name>
                <member_name>
                  <IDENTIFIER va="System"/>
                </member_name>
              </namespace_or_type_name>
            </namespace_name>
            <SEMICOLON va=";"/>
          </using_namespace_directive>
        </using_directive>
      </outer_declaration>
    </outer_declarations>
  </outer_declarations>
</compilation_unit>
```

Figure 7: An example of XML data for parse tree

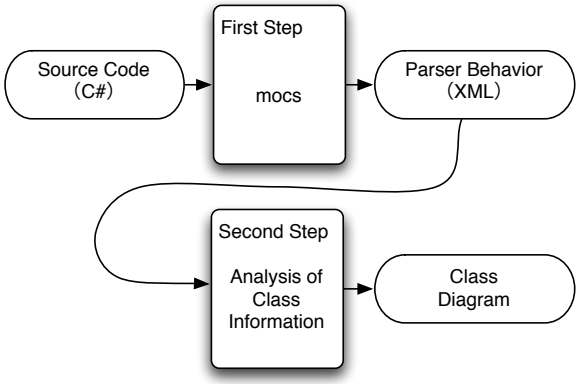


Figure 8: Two-steps for quick parser development

## 4 Tool Development Using XML Data

### 4.1 Quick Parser Development Using the Parser Behavior

It is possible to quickly develop the parser for reverse engineering tools using the parser behavior represented in XML. In this paper, the development approach is referred to as "two-step parsing." The parser is separated into two-steps for quick parser development, as shown in Figure 8. If we develop a reverse engineering tool for C#, mocs is available for the first step. It reads the C# source code and writes the parser behavior in XML. After the parser behavior is written in the first step, the parser behavior is read and class diagrams are generated in the second step.

As previously shown in Figure 2, MoJay reads the syntax rules G and generates a parser in the first step. Moreover, it generates lexical information and C# syntax rules H for the second step. When MoJay generates H, it removes the action codes in G to be invoked when the rules are recognized, and adds special symbols to identify each syntax rule.

A lexical analyzer in the second step sequentially reads the lexical information, as shown in Figure 9, from the parser behavior in XML, which is written by the first step. In the second step, it notifies lexical tokens and the timing for reduce actions to the parser.

The original parser generator Jay reads H, which is generated by MoJay in the first step, and then Jay generates a parser in the second step. After finishing the first step, all the lexical information and the matching order of syntax rules are already known so that the generated syntax rules cannot have any conflicts.

The parser in the second step does not build anything; instead it only checks the input from the viewpoint of the syntax rules. If we want to analyze class definitions and

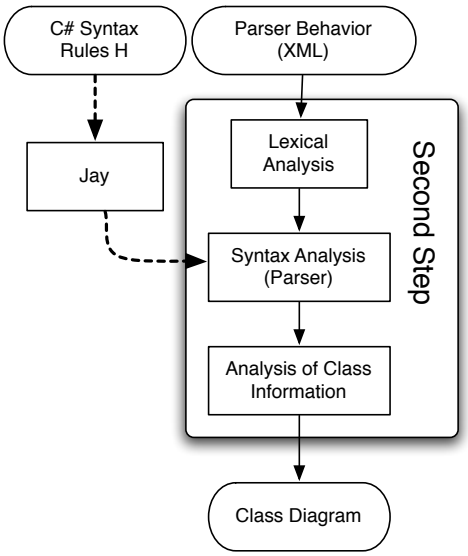


Figure 9: The second step for generating class diagrams

generate class diagrams, we need to embed appropriate functions in the syntax rules.

### 4.2 XML and Interoperability

The author developed a reverse engineering tool using two-step parsing. It reads C# source code and generates the class diagram that is shown in Figure 3. The tool was developed on Mac OS X using Mono. After completing the development of the production quality version, the source code was transferred to another PC (running on Windows XP), and we attempted to build the executable file using Cygwin[7] and Visual Studio 2005[8]. This building work was very simple, and it was carried out without any problems. This is because XML and C# function independently of all operating systems and computers.

Due to the interoperability of XML, we can implement the second step in different programming languages. The second step is developed by using the syntax rules generated by MoJay. The syntax rules for MoJay and Jay are compatible with Yacc and they are independent of all programming languages. This naturally indicates that mocs is used as the first step and the second step is implemented in another programming language (e.g. Java or C++).

XML documents utilize a lot of storage space to represent data that could be similarly modeled using a binary format or a simple text file format. This is because the XML documents are human-readable, platform-neutral, meta-data-enhanced, structured code. They can be 3 to 20 times as large as a comparable binary or alternate text file representation[9].

An experiment was designed to check the generated XML data size. In the case of a large file with 7,052 lines, the source code was translated to the parser behavior in XML with a size of 7M bytes. The size of the parser behavior exceeded that of the conventional XML document. However, storage cost is not a serious problem nowadays because the price of hard disk drives is increasingly becoming cheaper. Hence, even with a storage penalty, the quick parser development is a far superior option.

### 4.3 Application to SQL Parser

Two-step parsing is independent of the programming language and the platform. In order to clarify the independence, two-step parsing was applied to develop a SQL parser using PostgreSQL[10]. The conceptual structure of the query processor in PostgreSQL, as shown in Figure 10, is similar as one of the C# compiler.

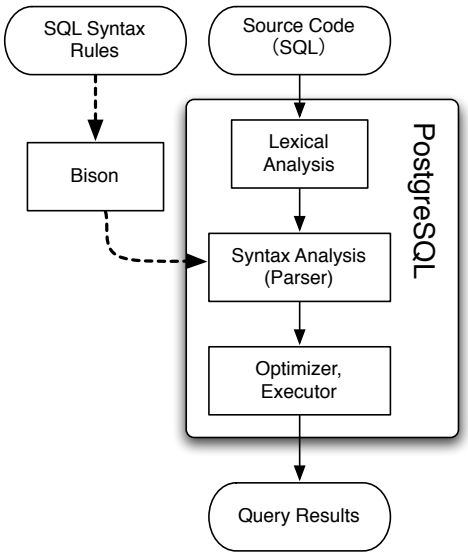


Figure 10: Conceptual structure of the query processor

There are some functions for lexical analysis, syntax analysis, optimizer, and executor which are implemented in the C programming language. Bison[11] is a parser generator to accept Yacc-compatible syntax rules. In the case of PostgreSQL, it reads more than 1,600 syntax rules for SQL and generates a parser written in C to analyze SQL source code.

The parser generator Bison used in the PostgreSQL is replaced with a modified parser generator MoBison. The input specification of MoBison is completely the same as one of Bison. MoBison reads the SQL syntax rules and generates the parser for the first step. Moreover, it statically generates lexical information and SQL syntax rules for the second step.

At run-time of the first step, the modified query processor in PostgreSQL analyzes the SQL source code, stores the

parser behavior as large objects and other information, and it returns the object id, as shown in Figure 11. The parser in the second step sends a query to the PostgreSQL in order to obtain the parser behavior generated in the first step.

The parser in the second step does not build anything; instead it only checks the input from the viewpoint of the syntax rules. In the author's experience, it took only one day to develop only the SQL parser by effectively using two-step parsing. If we want to analyze schema definitions and generate schema diagrams, we need more works to embed appropriate functions in the syntax rules, analyze relationship between table definitions, and draw schema diagrams.

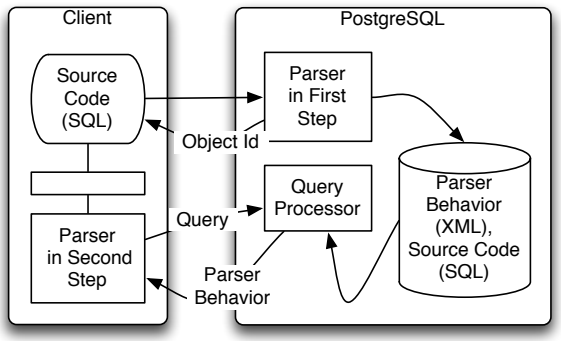


Figure 11: Two-step parsing for SQL parser

## 5 Concluding Remarks

This paper described XML externalization built into compiler front-ends and its application to quick reverse engineering tool development. A parser generator MoJay was developed to build XML externalization functionality into the compiler front-ends. After replacing an original parser generator with MoJay, generating a parser using it, and modifying a few lines of source code in the compiler, we were able to obtain a special compiler that generates three kinds of XML data, namely, lexical information, parser behavior, and parse tree.

The parser behavior was applied to quickly develop a reverse engineering tool for C#. During the tool development, a compiler front-end is separated into two-steps. In the first step, a special C# compiler mocs reads C# source code, analyzes it, and writes the parser behavior in XML. In the second step, the parser behavior in XML is read and analyzed. The reverse engineering tool shows a performance penalty from the viewpoint of the generated XML document size. However, even with a storage penalty, the quick development is a far superior option.

Free and open source software is rapidly proliferating in many fields. To accelerate this phenomenon, applications based on this paper must be extended to develop new technologies using XML.

## References

[1] Akmal B. Chaudhri, Awais Rashid, and Roberto Zicari ed., XML Data Management, Addison Wesley (2003).

[2] Ronald Bourret, XML and Databases, http://www.rpbourret.com/xml/XMLAndDatabases.htm.

[3] Main Page - Mono, http://www.mono-project.com/Main_Page.

[4] ECMA International, C# Language Specification, http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[5] Stephen C. Johnson. Yacc: Yet Another Compiler Compiler. In UNIX Programmer's Manual, Vol.2, pp. 353–387 (1979).

[6] Alfred V. Aho, Monica S. Lam, and et al., Compilers — Principles, Techniques, & Tools, Pearson Education (2006).

[7] Cygwin Information and Installation, http://www.cygwin.com/.

[8] Microsoft, Visual Studio 2005, http://msdn.microsoft.com/vstudio/.

[9] Zap Think, The "Pros and Cons" of XML, Zap Think Research Report (2001).

[10] *PostgreSQL*. http://www.postgresql.org/.

[11] The Free Software Foundation, *Bison - GNU parser generator*. http://www.gnu.org/software/bison/.