

# A JavaCC Parser for the UML-Based CDIF Transfer Format

Patrick Pagé

Rudolf K. Keller

Reinhard Schauer

Département IRO  
Université de Montréal  
C.P. 6128, succursale Centre-ville  
Montréal, Québec H3C 3J7, Canada  
{pagepa, keller, schauer}@iro.umontreal.ca

## 1 Introduction

As part of the project *SPOOL*, a joint university/industry collaboration between the University of Montreal and Bell Canada, we are developing a repository-based environment for design recovery and analysis in large-scale C++ telecommunications software [5, 9]. The purpose of this environment is to provide automated support for the extraction of design models from source code, by stepwise, human-controlled transformation of the source code into more abstract forms of representation. To use the functionality of other tools, such as source code parsers and metrics analysis tools, we needed a data exchange mechanism that would transfer the output of one tool into a format that could be understood by all other participating tools. Hence, we investigated several data interchange formats, including the *Rigi Standard Format* [12], the *Tuple Attribute Language* [3], the *Bell Intermediate Representation Language* [7], and the *Case Data Interchange Format (CDIF)* [1]. Among these alternative solutions, we opted for CDIF for its expressiveness and detailed specification.

The architecture of the SPOOL environment (Figure 1) consists of the C++ source code analysis system *GEN++* [2], the CDIF data transfer mechanism, a source code and design database, and a number of design recovery and analysis tools. We use *GEN++* to generate intermediate files for the relevant source code elements, such as classes, attributes and variables, operations and methods, free functions and operators, generalization relationships, friend relationships, function calls, and object instantiation. The purpose of this intermediate representation is to make the SPOOL tools independent of any specific programming language, and to provide a data exchange mechanism for Bell's tools for metrics analysis and clone detection, as well as for third-party tools. As the schema of our software repository is based on the UML metamodel 1.1 [10], we adapted the UML-

Compliant Interchange Format V1.0 [11] to reflect the UML 1.1 and our SPOOL-specific extensions due to our support of C++. In the following, we will refer to this format as UML-based CDIF Format, or more briefly, as UML-CDIF Format. The parser for this format, which we developed with Sun's Compiler Compiler *JavaCC* [4], generates a *Visitor* interface that allows tools to hook into the abstract syntax tree (AST) and pick out those data that are relevant for their specific functionality. By

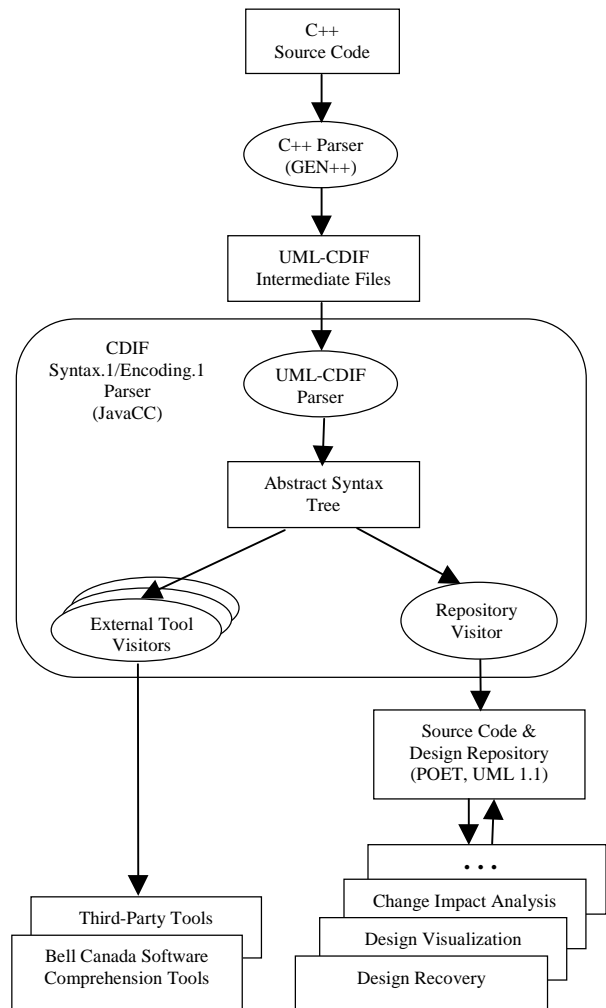


Figure 1: Architecture of the SPOOL environment.

This research was supported by the SPOOL project organized by CSER (Consortium for Software Engineering Research) which is funded by Bell Canada, NSERC (National Sciences and Research Council of Canada), and NRC (National Research Council of Canada).

implementing the Visitor interface, tools can, for example, transform UML-based CDIF into a different form of representation, store the data in tool-specific data structures, or directly perform some analysis task. For our purposes, we implemented the methods of the Visitor interface in order to store the source model represented as UML-based CDIF in our POET object repository [8].

The purpose of this paper is to introduce and discuss the parsing technology we are using for the CDIF Transfer Format. Section 2 details the JavaCC parser that we developed for CDIF. Section 3 explicates the modifications to the *CDIF SYNTAX.1/ENCODING.1* [1] based parser that are necessary to support *CDIF XML* [1]. Finally, section 4 rounds up the paper with some concluding remarks.

## 2 JavaCC-Generated CDIF Parser

The CDIF parser and import mechanisms of our environment are developed based on *Sun's* Java Compiler Compiler (*JavaCC*). JavaCC is an LL parser generator for Java, comparable to the well-known LR parser generator *yacc* [6] for C. Both JavaCC and the parsers it generates are certified as 100% pure Java and run on more than forty different platforms without any need for porting the code. This technology fits well in our all Java based SPOOL environment. In the following, we will walk through the different sections of the JavaCC grammar file we developed for supporting the UML-based CDIF Transfer Format.

The lexical analyser (token manager) of a JavaCC parser reads in the UML-based CDIF files, which, in our case, are generated with GEN++. It then groups the input characters to word-like tokens, and passes these to the syntax analyser (parser), which checks this stream of tokens for adherence to the CDIF grammar. The CDIF grammar file for JavaCC starts with the settings for the options offered by JavaCC (Figure 2).

```
Options {
  MULTI=true; VISITOR=true; STATIC=false; LOOKAHEAD=1;
}
```

Figure 2: JavaCC options.

The *main* compilation unit in a JavaCC grammar file is enclosed between `PARSER_BEGIN(cdifp)` and `PARSER_END(cdifp)`, where `cdifp` is used as the prefix for all generated Java classes and as the name for the generated main class (`javap.java`). The parser code that JavaCC generates is inserted immediately before the closing brace of the main compilation unit. In Figure 3, the main compilation unit opens the input file and calls `CDIFTransfer`, which is the root of the non-terminal symbols in our CDIF grammar. Then a *Visitor*

on the generated abstract syntax tree is constructed and attached to the `rootNode` to traverse the tree and perform some actions.

```
PARSER_BEGIN(cdifp)

Public class cdifp {

  public static void main(String args[]) throws ParseException {
    try {
      parser = new cdifp (new FileInputStream(args[0]));
    } catch (FileNotFoundException e) {
      System.out.println("\nFilename does not exist.\n");
      System.out.println("\nUsage: cdifp [filename]");
      System.exit(1);
    }

    ASTCdifTranfer rootNode      = parser.CDIFTransfer();

    cdifpDatabaseVisitor visitor = new cdifpDatabaseVisitor();
    rootNode.jjtAccept(visitor, null);
  }
}

PARSER_END(cdifp)
```

Figure 3: JavaCC main compilation unit.

The main compilation unit is followed by the list of terminal symbols (Figure 4), which might include a `SKIP` region indicating those symbols which the token manager is to ignore. The subsequent `TOKEN` sections are used to specify the terminal symbols as defined by CDIF and the UML. A powerful feature of JavaCC is the possibility to specify fairly complex regular expressions, as illustrated in our example with `INTEGER`, which then can be referred to anywhere else in the grammar.

```
SKIP : // tokens to ignore
{
  " " | "\t" | "\n" | "\r"
}

TOKEN: // identifiers
{
  <INTEGER      : ( <DIGIT> )+ >
  | <DIGIT      : ["0" - "9"] >
}

TOKEN : // CDIF tokens
{
  <C_OPEN_SCOPE   : "<">
  | <C_CLOSE_SCOPE : ">">
  | <C_DOT         : ".">
  | <C_COLON       : ":">
  | <C_HEADER      : "HEADER">
  | <C_SUMMARY_KEYW : "SUMMARY">
  etc.
}

TOKEN : // UML tokens
{
  <U_ELEMENT      : "Element">
  | <U_OPERATION   : "Operation">
  | <U_METHOD      : "Method">
  | <U_BEHV_FEATURE : "BehavioralFeature">
  | <U_MODEL       : "Model">
  | <U_VISIBILITY_KIND : "public" | "protected" | "private">
  | <U_AGGREGATION_KIND : "none" | "aggregate" | "composite">
  etc.
}
```

Figure 4: JavaCC token section.

The subsequent section contains the list of production rules, which are expressed either as Java code or in Backus-Naur Form (*BNF*). Java code may be applied for non-context free parts of the grammar. This remaining

section consists only of BNF production rules. The left-hand side of a BNF production is a declaration of a non-terminal symbol, which is written like a method declaration in Java. JavaCC generates for each non-terminal an identically named method in the parser class (*cdifp.java*). Parameters and return values can be declared to pass values up and down the parse tree. The right hand side of a BNF production rule starts with a set of Java declarations and code, which is generated into the beginning of the method, and thus carried out every time this non-terminal is used. The subsequent expansion unit, or parser actions, of the non-terminals instruct the generated parser on how to make choices. It is enclosed within braces and can consist of any number of Java declarations and code. The expansion unit can also include a local *lookahead*, specified either as a lookahead limit (to limit the maximum number of tokens of lookahead that may be used for choice determination purposes), a syntactic lookahead (to test the input stream against a regular expressions), or a semantic lookahead (to test the tokens of the input stream with a boolean expression). Lexical and syntactical analysis errors can be caught using standard Java exception handling. The lexical analyzer throws *TokenMgrError* and the syntax analyser *ParseException*. Reporting and recovery code can be inserted into the *catch* clauses. Figure 5 illustrates three examples of production rules for the CDIF *HeaderSection*, the CDIF *SummarySection*, and the UML *GeneralizableElement*.

```

void HeaderSection() :
{
  <C_OPEN_SCOPE> <C_HEADER>
  SummarySection()
  <C_CLOSE_SCOPE>
}

void SummarySection() :
{
  try {
    <C_OPEN_SCOPE> <C_SUMMARY>
    <C_CLOSE_SCOPE>
  } catch (ParseException e) {
    System.out.println("Fatal Error in Summary Section!");
  }
}

void UMLGeneralizableElementMetaClassAttrib() :
{ int type;
  Token value;
}
{
  UMLNamespaceMetaClassAttrib()
  | ( ( <U_ISROOT > { type=U_ISROOT; } val=<U_BOOL>)
    | <U_ISLEAF > { type=U_ISLEAF; } val=<U_BOOL>)
    | <U_ISABSTRACT > { type=U_ISABSTRACT; } val=<U_BOOL>)
  )
}
}

```

Figure 5: JavaCC production rules.

An add-on to JavaCC, *JJTree* is a pre-processor for JavaCC that inserts AST (abstract syntax tree) building actions into the JavaCC source code. By default, *JJTree* generates a class for each non-terminal symbol in the grammar. This standard behavior can be changed so that,

for example, classes be generated only for certain non-terminals or additional nodes be added into the tree. Most important to tools that need to read and process only certain data, *JJTree* generates a Visitor interface with an operation for each type of AST node. We implemented a Visitor with methods to fill our design database with the source code represented by the UML-based CDIF intermediate file.

Our grammar file for the UML-based CDIF format, supporting simple error handling, consists of 1,763 lines of JavaCC code. More encompassing error handling that automatically pops up a window showing the erroneous intermediate code is under construction. A graphic front end that lets the user specify what kind of data to extract from the intermediate files and to which form the output is to be converted is on our to-do list.

### 3 XML Support

With the Extensible Markup Language (XML) becoming the emerging standard for data interchange over the Internet, the CDIF Technical Committee has proposed the CDIF XML-based Transfer Format [1] as an alternative to the well-known SYNTAX.1/ENCODING.1-based transfer format, which we used in the initial phase of our work. It adapts SYNTAX.1/ENCODING.1 to be fully compliant with the XML specifications. In this section, we will briefly discuss some issues involved in providing support for XML.

Adapting our CDIF SYNTAX.1/ENCODING.1 parser into a CDIF XML parser is a relatively straightforward task. Since only the syntax has changed, but not the semantics of the production rules, the AST is still the same. Hence, the operations on the AST do not have to be changed, and the JavaCC generated Visitor methods, which perform tool-specific actions on the AST-nodes, are not affected by such a transition.

When changing the encoding form CDIF ENCODING.1 to CDIF XML, we first need to modify the set of tokens to reflect the XML. There are very few that need to be changed, among these “(”, “)”, “:HEADER”, “:MODEL”, “#|” and “|#”. The major, but still straightforward task of the parser adaptation is to change the syntax of the right-hand sides of the production rules to XML, which have the same content, but a different structure. Figure 6 illustrates the syntactical differences between CDIF SYNTAX.1/ENCODING.1 and CDIF XML. For more detailed information, we refer to the CDIF documentation set [1].

```

CDIF SYNTAX.1/ENCODING.1 :
(Method 1
  (Name "myMethod")
)
(Class 2
  (Name "myClass")
)
(Classifier.feature.Feature 3 2 1)

CDIF XML :
<ME   Name="Method"
      ID="1">
  <MA   Name="Name"
        Value="myMethod"
  />
</ME>
<ME   Name="Class"
      ID="2">
  <MA   Name="Name"
        Value="myClass"
  />
</ME>
<MR   Name="Classifier.feature.Feature"
      ID="3"
      Src="|1"
      Dest="|2">
</MR>

```

Figure 6: SYNTAX.1/ENCODING.1 vs. XML.

## 4 Conclusion

We have developed a JavaCC parser for the UML-based CDIF SYNTAX.1/ENCODING.1 Transfer Format, and are currently in the process of adapting the parser to support CDIF XML. The primary purpose for this data transfer mechanism is to provide an interface to GEN++, the C++ source code analysis system adopted in the SPOOL environment. We use GEN++ to parse C++ source code and generate an intermediate representation of all the programming constructs required for design recovery and analysis of design quality.

JavaCC is a robust and flexible parser generator that allows different tools to process the parsed input stream in different ways. JavaCC is seamlessly integrated into Java and thus provides all the possibilities of Java for customizing the JavaCC default parser generation capabilities to fit the requirements at hand. Most noteworthy, by implementing the methods of the generated Visitor interface, different tools can hook into the AST and perform arbitrary computations on the syntax tree, without the necessity of re-generating the parser.

Our parser was developed as a student term project in a four person-months effort, which included studying all the technology involved in this project, such as GEN++, JavaCC, and the Poet database management system. The parser is available from the authors of this paper on request. Based on our experience, we can highly recommend both the CDIF Transfer Format and JavaCC to tool builders that need flexible and robust technology

to let multiple, independent tools, with different data needs and different input conventions, collaborate on some overall task.

## 5 References

- [1] CDIF Transfer Format. *Online documentation set*, 1997. CDIF Technical Committee, Electronic Industries Association, Arlington, VA. Available at <<http://www.cdif.org>>.
- [2] Premkumar T. Devanbu. GENOA - a customisable, language- and front-end independent code analyser. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307-317, Melbourne, Australia, 1992.
- [3] Ric C. Holt. *An Introduction to TA: The Tuple-Attribute Language*. Available at <<http://turing.toronto.edu/~holt/papers/ta.html>>.
- [4] Java Compiler Compiler: The Java Parser Generator. *Online documentation for version 0.7.1*. Sun Microsystems, Palo Alto, CA. Available at <<http://www.sun.com/suntest/JavaCC>>.
- [5] Rudolf K. Keller and Reinhard Schauer. Design components: Towards software composition at the design level. In *Proceedings of the 20th International Conference of Software Engineering*, pages 302-311, Kyoto, Japan, April 1998.
- [6] Tony Mason and Doug Brown. *Lex and yacc*. O'Reilly, Sebastopol, CA, 1990.
- [7] Jean Mayrand and Ettore Merlo. Évaluation de la qualité des logiciels. École Polytechnique de Montréal. Technical Report, Montréal, QC, Canada January 1996. In French.
- [8] POET Java ODMG Binding. *Online documentation*, September 1997. POET Software Corporation, San Mateo, CA. Available at <<http://www.poet.com/>>.
- [9] Reinhard Schauer and Rudolf Keller. Pattern visualisation for software comprehension. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 4-12, Ischia, Italy, June 1998.
- [10] UML. *Documentation set version 1.1*, September 1997. Rational Software Corporation, Santa Clara, CA. Available at <<http://www.rational.com/>>.
- [11] UML-Compliant Interchange Format. *Version 1.0*, January 1997. Rational Software Corporation, Santa Clara, CA.
- [12] Kenny Wong. Rigi User Manual. Available at <<http://www.rigi.csc.uvic.ca/rigi/manual/org/user.html>>.