Design and Implementation of a Java to CodeML converter

Rabin Vincent

`<r.vincent@iu-bremen.de>`

Supervisor: Michael Kohlhase

Guided Research Spring 2005

International University Bremen

Abstract

Code Markup Language (CodeML) defines an XML-based language for capturing the syntax and semantics of program code. This project has created a program which is able to automatically convert source code in the Java programming language into a CodeML document. This paper describes a subset of the CodeML language, and the construction and working of the converter program.

Contents

Chapter 1

Introduction

1.1 Program Code Representation

Program code has two purposes, the primary being a method of instructing the machine to perform actions desired by the programmer. The second, auxiliary purpose of program code is to communicate about algorithms and programs between humans [1].

Program code is widely exchanged, on the Internet for example, but there is currently no standard format for the representation of program code. Most source code is exchanged as plain text files.

Although this format is convenient, it provides no added value such as aids to the reader to understand the content. Features such as hyper-linking between parts of the code, section folding, and syntax highlighting are tied to a specific development environment and there is no way to transfer this information easily between users and environments. Each editor has to implement a parser for each of the languages it needs to display and annotate.

1.2 Introducing CodeML

Code Markup Language, CodeML, is an XML-based format to represent program code, and denote both its structure and content. The goal of CodeML is to enable program code to be served, received and processed on the World Wide Web, just as HTML has enabled this functionality for text [1].

CodeML is programming language neutral. Editors can display and understand several different programming languages by simply supporting the CodeML syntax. Further more, this document can be exchanged without loss of information, and users can be sure that other persons viewing the document would have the same information.

Being XML-based, CodeML has the advantage of being easily processed and handled using technologies such as XSL, XQuery, XPath and CSS.

1.3 Creating CodeML documents

In order to convert source code written in a specific language into CodeML, it is not feasible to perform the conversion manually. This conversion must be done automatically by the editor or the integrated development environment that is being used.

Currently, no program capable of doing this exists. Therefore, in this project, this project has created a program, **Java2CodeML**, to translate Java program code into CodeML. This program serves as a proof-of-concept implementation of the CodeML syntax and also provides a mechanism to easily generate CodeML documents from a corpus of source code in order to verify and improve the specification itself.

Chapter 2

CodeML Syntax

The section describes the syntax of CodeML, as defined in [1]. Additionally, during the course of development of `Java2CodeML`, some small additions were made, and these are also noted.

A CodeML document consists of two cross-referenced sections, a presentation section and a content section. The presentation section captures the syntactic structure of the program code and the content section the semantic information.

Since this project aimed to only work with the syntactic sections, only those will be discussed. Information on the semantic section and the interaction of the two can be found in [1].

2.1 Presentation Markup

The *presentation markup* (henceforth simply *presentation*) classifies sections of program code according to their syntactic function, and provides sufficient information for the program code to be reconstructed accurately from the CodeML.

The presentation is enclosed within a `pcode` element. The `pcode` element has a `format` attribute which indicates the programming language the code fragment is in.

2.1.1 Elements

The presentation uses the following XML elements.

cpb

The `cpb` elements are meant to be used for basic types. It has an optional attribute `type` which classifies the contents of the element. According to the CodeML specification, type can take the value 'number'. We also introduce another possible value for type, 'string' which will be used for strings and character sequences.

Listing 2.1: Example of <cpb>

```
<cpb type="number">42</cpb>
<cpb type="string">"Java"</cpb>
```

cpi

cpi is used for variable names.

Listing 2.2: Example of <cpi>

```
<cpi>i</cpi>
<cpi>count</cpi>
```

cptype

cptype is an element to hold data types.

Listing 2.3: Example of <cptype>

```
<cptype>int</cptype>
<cptype>boolean</cptype>
```

cpo

The cpo element is used for built-in operators of the programming language and functions/methods used in the program. It has an optional type attribute which can take on of the following values: 'built-in', 'imported' and 'defined'.

Listing 2.4: Example of <cpo>

```
<cpo type="built-in">&lt;</cpo>
<cpo type="imported">System.out.println</cpo>
```

cpg

The cpg element is used to group program code fragments. It has the following attributes, all of them optional.

- open and close specify the opening and closing parenthesis for the block

- `separator` specifies the separator to be used in between children of this group. The default is a space.
- `indent` refers to the indentation level the group should be at
- `breakO` and `Obreak` denote the type of line break before and after the opening parenthesis
- `breakC` and `Cbreak` denote the type of line break before and after the closing parenthesis
- `breakS` and `Sbreak` denote the type of line break before and after separators

Listing 2.5: Example of `<cpg>`

```
<cpg open="{" close="}" Obreak="hard"> ... <cpg>
<cpg close=";" Cbreak="hard"> ... </cpg>
```

2.2 Raw Code

The raw code section simply includes a copy of the original source code within a `rawcode` element. This element has an attribute, `format` which specifies the programming language of the code included. Since the program code needs to be printed verbatim, it is enclosed in a CDATA block to prevent being parsed by XML parsers.

Listing 2.6: Example of raw code

```
<rawcode format="python"><![CDATA [
def function(n):
        if n == 5:
                return True
        return False
]]>
</rawcode>
```

2.3 Complete document

The root of the XML document is the `code` element. This element has an attribute, `format` which indicates the type of CodeML representation enclosed within (usually `'multi'`, for *multiple*). Inside this `code` element is the `semantics` element.

Inside the `semantics` element, the different sections are present. The first section is content markup (which is not shown here) and the rest are presentation sections. Listing 2.7 shows the CodeML representation of a simple C code fragment.

Listing 2.7: A simple CodeML document

```
<code format="multi">
        <semantics>
                <pcode format="c">
                        <cpg close=";" Cbreak="hard">
                                <cptype>int</cptype>
                                <cpi>i</cpi>
                                <cpo type="built-in">=</cpi>
                                <cpb type="number">5</cpb>
                        </cpg>
                </pcode>
                <rawcode format="c"><![CDATA [
int i = 5;
]]>
                </rawcode>
        </semantics>
</code>
```

Chapter 3

Implementation of `Java2CodeML`

3.1 Background information

Translating source code from one form to another involves the following processes.

3.1.1 Lexical and Syntactic Analysis

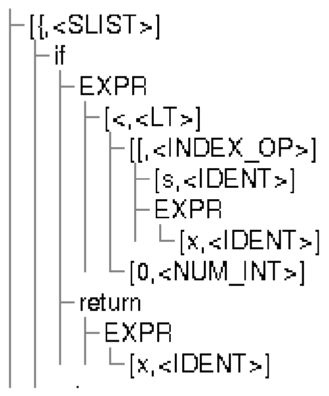
The source file is streamed to a *lexical analyzer* (or *lexer*) which quantifies the stream of characters into discrete groups that, when parsed by the parser, have meaning. These groups are called tokens, and they are the components of the programming language such as keywords, operators, and identifiers. [2]

Each individual token has some semantic worth, but the lexer does not consider the semantic worth within the context of the program or the segment of code. The *parser* (or *syntactical analyzer*) receives this stream of tokens from the lexer, and checks if the tokens confirm to the syntax of a language as defined by the grammar of the language. [2]

3.1.2 ASTs and Tree walking

For complex problems, There is the need for the construction of a tree, called an *abstract syntax tree* (AST) from the sequences of tokens recognized by the parser. When this tree is built, a complete structured representation of the program code is available.

To do something useful with the AST, a tree walker can be used. A tree walker is an algorithm which starts from the root of a fully-built AST, and walks down the nodes of the tree, performing actions whenever required. The tree walker is the heart of the program and performs the actual translation.



```
├[{,<SLIST>]
│├if
││├EXPR
│││├[<,<LT>]
││││├[[,<INDEX_OP>]
│││││├[s,<IDENT>]
│││││├EXPR
││││││└[x,<IDENT>]
││││└[0,<NUM_INT>]
││├return
│││├EXPR
││││└[x,<IDENT>]
```

Figure 3.1: Section of an ANTLR AST for a Java program

3.1.3 Parser Generators

Generally, parsers, lexers and tree builders are not build by hand. There exist programs, called *parser generators* or *compiler compilers* which are able to generate lexers and parsers automatically. These programs use as input a *grammar*, a description of the program to be parsed as a context-free grammar. This defines the rules the lexer uses to recognize tokens and the rules the parser uses to group them.

3.2 Available resources

3.2.1 The ANTLR parser generator

ANTLR, ANother Tool for Language Recognition, provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions. It can create lexers, parsers, and tree builders [4]. ANTLR is written in Java and is in the public domain.

ANTLR has its own syntax for grammar, and there are a wide variety of publicly available grammars for various languages and formats.

3.2.2 Grammar for the Java language

ANTLR ships with a complete grammar which is capable of recognizing Java 1.3 [3], written by Terence Parr et al. Although grammars for newer revisions of the Java language are available, it was chosen to use this grammar since it has been around for longer and has such undergone several improvements. Also, this grammar comes with an ANTLR tree grammar which generates an AST of the Java source code.

3.2.3 JavaEmitter

JavaEmitter is a program that generates Java code from an ANTLR AST [5]. This open-source program contains code for a complete tree walker for the Java AST generated by the above mentioned tree grammar. The code for the `Java2CodeML` is based partly on JavaEmitter.

3.3 Structure of `Java2CodeML`

`Java2CodeML` is written in Java, and is operated through a command line interface. It consists of a main program class, `Java2CodeML` which simply provides the interface to the program, and runs the converter classes. The real work is done by two classes, `Presentation` and `RawCode`. `RawCode` is straightforward, and simply places the original program file in a `CodeML` element, as specified in 2.2.

The `Presentation` class performs the parsing and translation of the source code.

3.4 Working of the `Presentation` translation

The program first loads the Java source file to be converted and runs the Java lexer, parser and tree builder on it to create the `AST`. Then a tree walker is run over the `AST`. The tree walker consists of recursive calls to a `print` method, which contains a large `switch` statement, that performs actions depending on the type of the node currently being worked on.

3.4.1 Example: Conversion of a `for` loop

Listing 3.1 shows a part of a Java source code AST. There is a node `LITERAL_for` with four children, and each of these have children, and so on.

Listing 3.1: Section of AST for a `for` loop

```
LITERAL_for --- FOR_INIT -- ELIST -- EXPR -- ...
            |-- FOR_CONDITION -- EXPR -- ..
            |-- FOR_ITERATOR -- ELIST -- EXPR -- ...
            |-- SLIST -- ...
```

The following listing shows part of the code within the tree walker, which is contained in the `print` method. This is the `case` which handles the `for` loop element shown above. `printCpg`, `printNode`, `printStartNode`, and `printEndNode` are functions which output the CodeML elements. `getChild` returns the first child of the current node of the specified type. (For the full source code, see Appendix A).

Listing 3.2: Part of the tree walker

```
case LITERAL_for:
        printNode(cpo_builtin, "for");

        printCpg("(", ")", "Cbreak=\"hard\"");

        printCpg("", ";");
        print(getChild(ast, FOR_INIT));
        printEndNode(cpg);

        printCpg("", ";");
        print(getChild(ast, FOR_CONDITION));
        printEndNode(cpg);

        printStartNode(cpg);
        print(getChild(ast, FOR_ITERATOR));
        printEndNode(cpg);

        printEndNode(cpg);

        print(getChild(ast, SLIST));
        break;

case FOR_INIT:
case FOR_CONDITION:
case FOR_ITERATOR:
        print(child1);
        break;
```

The above statements, in concert with the rest of the `case`s which handle the inner elements, will produce a CodeML representation in the following fashion:

Listing 3.3: Example for loop CodeML

```
<cpo type="built-in">for</cpo>
<cpg open="(" close=")" Cbreak="hard">
        <cpg close=";">
                ...
        </cpg>
        <cpg close=";">
                ...
        </cpg>
        <cpg>
                ...
        </cpg>
</cpg>
...
```

Chapter 4

Results

The completed `Java2CodeML` is capable of generating CodeML presentation markup for every construct in a Java program, according to the current CodeML specification.

4.1 Example generated CodeML

As an example of the capabilities of the `Java2CodeML`, the following is the `CodeML` representation of a Java program (which is show in the `rawcode` section) generated by `Java2CodeML`.

Listing 4.1: CodeML document

```
<?xml version="1.0"?>
<code type="multi">
  <semantics>
    <pcode format="java">
      <cpg>
        <cpg close=";" Cbreak="hard">
          <cpo type="built-in">import</cpo>
          <cpo type="imported">java.io.*</cpo>
        </cpg>
        <cpg>
          <cpo type="built-in">class</cpo>
          <cpo type="defined">LongTest</cpo>
        </cpg>
        <cpg open="{" close="}" breakO="hard">
          <cpg>
            <cpg>
              <cpo type="built-in">public</cpo>
              <cptype>boolean</cptype>
              <cpo type="defined">check</cpo>
              <cpg open="(" close=")">
                <cpg>
                  <cptype>int</cptype>
                  <cpi>a</cpi>
    </cpg>
  </cpg>
</cpg>
<cpg open="{" close="}" breakO="hard">
  <cpg close=";" Cbreak="hard">
    <cptype>int</cptype>
    <cpi>res</cpi>
    <cpo type="built-in">=</cpo>
    <cpb type="number">0</cpb>
  </cpg>
  <cpo type="built-in">for</cpo>
  <cpg open="(" close=")">
    <cpg close=";">
      <cptype>int</cptype>
      <cpi>i</cpi>
      <cpo type="built-in">=</cpo>
      <cpb type="number">0</cpb>
    </cpg>
    <cpg close=";">
      <cpi>i</cpi>
      <cpo type="built-in">&lt;</cpo>
      <cpi>a</cpi>
    </cpg>
    <cpg>
      <cpi>i</cpi>
      <cpo type="built-in">++</cpo>
    </cpg>
  </cpg>
  <cpg open="{" close="}" breakO="hard">
    <cpg close=";" Cbreak="hard">
      <cpi>res</cpi>
      <cpo type="built-in">++</cpo>
    </cpg>
  </cpg>
  <cpo type="built-in">switch</cpo>
  <cpg open="(" close=")">
    <cpi>res</cpi>
  </cpg>
  <cpg open="{" close="}" Obreak="hard">
    <cpg close=":" Cbreak="hard">
      <cpo type="built-in">case</cpo>
      <cpb type="number">5</cpb>
    </cpg>
    <cpg open="{" close="}" breakO="hard">
      <cpg close=";" Cbreak="hard">
        <cpo type="built-in">return</cpo>
        <cpo type="built-in">true</cpo>
      </cpg>
    </cpg>
    <cpg close=":" Cbreak="hard">
      <cpo type="built-in">default</cpo>
    </cpg>
    <cpg open="{" close="}" breakO="hard">
      <cpg close=";" Cbreak="hard">
        <cpo type="built-in">return</cpo>
```

```
            <cpo type="built-in">false</cpo>
          </cpg>
        </cpg>
      </cpg>
    </cpg>
  </cpg>
  <cpg>
    <cpg>
      <cpo type="built-in">public</cpo>
      <cpo type="built-in">static</cpo>
      <cptype>void</cptype>
      <cpo type="defined">main</cpo>
      <cpg open="(" close=")">
        <cpg>
          <cptype>String[]</cptype>
          <cpi>args</cpi>
        </cpg>
      </cpg>
    </cpg>
    <cpg open="{" close="}" breakO="hard">
      <cpg close=";" Cbreak="hard">
        <cptype>LongTest</cptype>
        <cpi>lt</cpi>
        <cpo type="built-in">=</cpo>
        <cpo type="built-in">new</cpo>
        <cpo>LongTest</cpo>
        <cpg open="(" close=")"/>
      </cpg>
      <cpg close=";" Cbreak="hard">
        <cptype>int</cptype>
        <cpi>a</cpi>
        <cpo type="built-in">=</cpo>
        <cpb type="number">5</cpb>
      </cpg>
      <cpo>if</cpo>
      <cpg open="(" close=")">
        <cpo>lt.check</cpo>
        <cpg open="(" close=")">
          <cpi>a</cpi>
        </cpg>
        <cpo type="built-in">==</cpo>
        <cpo type="built-in">true</cpo>
      </cpg>
      <cpg open="{" close="}" breakO="hard">
        <cpg close=";" Cbreak="hard">
          <cpo>System.out.println</cpo>
          <cpg open="(" close=")">
            <cpb type="string">"Success!"</cpb>
          </cpg>
        </cpg>
      </cpg>
    </cpg>
  </cpg>
 </cpg>
</cpg>
```

```
    </pcode>
    <rawcode format="java"><![CDATA[
import java.io.*;

class LongTest {
        public boolean check (int a) {
                int res = 0;
                for (int i = 0; i < a; i++) {
                        res++;
                }

                switch (res) {
                        case 5:
                                return true;

                        default:
                                return false;
                }
        }

        public static void main (String[] args) {
                LongTest lt = new LongTest();
                int a = 5;

                if (lt.check(a) == true) {
                        System.out.println("Success!");
                }
        }
}
]]></rawcode>
  </semantics>
</code>
```

As can be seen, `Java2CodeML` effectively handles several kinds of Java constructs and converts them to valid CodeML.

4.2 Limitations

Although `Java2CodeML` can handle complete Java programs, it has the following limitations:

- Cannot distinguish between `imported` and `defined` methods, since this cannot be done just by parsing. This requires use of the Java *Reflection* classes (see Section 4.3, Future Work)
- Can only parse Java source files which have a class defined. In other words, it cannot markup code snippets. If snippets need to be converted to `CodeML`, they will have to be enclosed in a class (and method, if necessary) for the program to be able to convert them.

For the program to be able to convert snippets, code must be added for it to recognize the level at which the snippet is and then start parsing from that rule. Currently, the parser simply starts parsing from the main rule, which defines a `compilationUnit`, a Java source file.

4.3 Future work

Apart from work to iron out the limitations (see above) of the `Java2CodeML` implementation, future work done should ideally be on the generation of the CodeML content markup. In the CodeML specification, this would require a more complete definition of the types assigned to functions and built-ins in the Java language. On the implementation of the program, the program will require the use of Java's *Reflection* classes to introspect the source code being worked on.

After the content markup is finished, the important, and perhaps most challenging, part is the cross-referencing of the presentation markup that is available now and the content markup.

4.4 Conclusion

With the help of publicly available programming tools, `Java2CodeML` is able to efficiently translate Java source code into language-neutral CodeML. The output produced by `Java2CodeML` is used to debug and improve the CodeML specification.

Bibliography

[1] Kohlhase, Michael. CodeML: An Open Markup Format for the Content and Presentation of Program Code.

[2] Mill, Ashley J.S. ANTLR. `http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/antlr/antlr.html` (accessed April 24, 2005).

[3] Parr, Terence et al. Java 1.3 Recognizer Grammar and Java 1.3 AST Recognizer Grammar. `http://www.antlr.org/grammar/java` (accessed April 23, 2005).

[4] Parr, Terence. ANTLR Parser Generator. `http://www.antlr.org` (accessed April 23, 2005).

[5] Tripp, Andy. JavaEmitter. `http://jazillian.com/antlr/emitter.html` (accessed April 23, 2005).

Appendix A

Source code

Due to the fact that the source code runs into tens of pages, it is not included here. `Java2CodeML`, its complete source code, and the components and grammars it uses can be obtained from:

`http://pandora.iu-bremen.de/~rvincent/codeml/`