Parallel Parsing-based
Software Engineering

Ravi Khadgi, Reeta Rai, Stefan Andrei

No. 1, March 2010
Parallel Parsing-based Software Engineering

Ravi Khadgi
Lamar University
Computer Science Department
er.ravi@gmail.com

Reeta Rai
Lamar University
Computer Science Department
reeta.rai@gmail.com

Stefan Andrei
Lamar University
Computer Science Department
sandrei@my.lamar.edu

March 31, 2010

Abstract

In most of large software engineering projects, we often are in possession of the source code of an application, but we miss its programmer, the design, and the documentation. In order to understand these software projects, we need reverse engineering tools useful because they offer more semantics about the implementation. Modern software engineering projects require a large number of developers. These projects have a high complexity of interaction between components, run on multiple hardware and software platforms, and hence include large quantities of source code. Provided a multi-core processor, which are found in almost all computers these days, we could use parallel programming techniques to efficiently process such large projects. Programming languages have evolved over the years in order to cover higher level programming concepts. It is obvious that we might want projects written in a particular programming language to be rewritten on another language. Of course it is desirable to achieve this goal without changing the original design. In this case, we could apply the reverse software engineering technique to source code written in one programming language. As a result, we will obtain a representation of the original design. After that, we could generate the source code in a new language through the process of forward software engineering techniques. This project uses a parallel parsing technique to convert a source code, which is written in an object-oriented programming language, into UML diagrams and vice versa.

1 Introduction

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems [15]. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. It simplifies the complex process of software design, making a detailed plan for construction. The purpose of modeling is developing a model for an industrial-strength software system prior to its construction or renovation [3]. UML does not guarantee project success but it does improve many important things. For example, it significantly lowers the perpetual cost of training and retooling when changing between projects or organizations. It provides the opportunity for new integration between tools, processes, and domains.

UML was developed jointly by Grady Booch, Ivar Jacobson, and Jim Rumbaugh at Rational Software Corporation, with contributions from other leading methodologists, software vendors, and many users [8]. Based on extensive use of the Booch and Jacobson methods, UML is the evolution of these approaches to business process, object, and component modeling. UML provides the application modeling language for: business process modeling with use cases, class and object modeling, component modeling, distribution and deployment modeling. The Rational Software Corporation provides a professional software tool called the Rational Rose Enterprise

Parsing, also known as syntactic analysis, is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar. Hence, parsing is a process of determining how a string of an arbitrary number of terminals can be generated by a finite grammar.

There are basically two types of parsing: the top-down parsing and bottom-up parsing. The classification is done in order in which the nodes in the parse tree are constructed. In top-down parsers, construction starts at the root and proceeds towards the leaves whereas in bottom-up parsers, construction starts at leaves and proceed towards the root [1]. LL (read input from Left to right and produce Leftmost derivation) parsers and recursive-descent parser are examples of top-down parsers. Bottom-up parsers include LR (read input from Left to right and produce Rightmost derivation) parsers.

Object-oriented software engineering is a discipline that utilizes the object-oriented paradigm to achieve the aims of software engineering. The object-oriented paradigm basically deals with objects. An object is a data structure that has two characteristics: state and behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (or functions) [4]. An object can also be viewed as a software component that incorporates both data and the actions that are performed on that data. For example, in a bank account, data can be account balance whereas actions can be deposit or withdrawal of balances. The main features of the object oriented paradigm are polymorphism, encapsulation, and inheritance.

Forward engineering is the traditional process of moving from high-level abstractions and logical implementation-independent designs to the physical implementation of a system. It steps upward from architectural requirements to designing its implementation. However, certain system aspects may be implemented directly from an architectural description [7].

Reverse engineering analyses a system, identifies its components and their relationships, and creates a representation of the system in another form or at a higher level of abstraction. Reverse engineering can also be defined as the process of extracting software system information, including documentation, from a source code. It deals with the extraction of design artifacts and synthesizing abstractions that are less implementation-dependent. Reverse engineering is important because it assists in redesigning the system for better maintainability or in producing a copy of a system without access to the design from which it was originally produced. But, this does not mean it is a process of change or replication. It is a process of examinations [7].

Traditionally, most programs are written sequentially. There are a few parallel implementations of these sequential programs using complex methods and that run on expensive computers, mainly supercomputers. The trend, however, is shifting towards achieving parallelism at a low cost. Several parallel programming languages, libraries and environments have been developed to ease the task of writing programs for multiprocessors. However, Parallel computer programs are more difficult to write than sequential ones because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common [14].
The parallel implementation is based on either the message passing or the shared memory model. The standard programming interface for the message passing model is MPI (Message Passing Interface) [12], offering a complete set of communication routines. Likewise, OpenMP (Open Multi-Processing) [5] is the standard for directive-based shared memory programming. Shared memory and message passing concurrency have different performance characteristics. The per-process memory overhead and task switching overhead is lower in a message passing system. However, it has greater overhead of message passing.

1.1 Motivation

It would be very difficult to understand the structure of large projects without having their design and documentation. The original software developer team who has the knowledge of the entire project may not be any longer available. It is also possible that the design might have been lost or was never created. We will only be able to get little if any information just by looking at the source code of such projects.

Another case where design plays a vital role is rewriting of the code in different programming languages. Suppose we have a project written in one programming language, for example C++, for which there is no sufficient design and documentation. We want to rewrite the project in another programming language, for example Java. It would be great if we are able to generate the design of the project from the previous source code. As a result, we could rewrite the source code in a programming language different from the original language.

In such cases, an automated tool that can convert source code into a proper graphical representation (such as UML class diagram) and vice versa would be very handy. In addition, we could use parallel programming techniques to process large projects.

To the best of our knowledge, there is no parallel implementation of a reverse software engineering tool.

1.2 Problem Description

Reverse software engineering (Figure 1-1) is exactly the opposite of forward software engineering (Figure 1-2). Forward software engineering is the generation of source code from a given specification. It is a sequence of going from requirement to designing the implementations. In reverse engineering, the UML tool loads all the files of the application/system, identifies dependencies between the various classes, and essentially reconstructs the entire application structure along with all the relationships between the classes. Reverse engineering is a feature normally provided by sophisticated high-end UML tools. Reverse engineering has many motivations. For instance, reverse engineering is done because the documentation of a particular device has been lost or was never written. Reverse engineering is also useful to examine how a product works and to estimate costs of components.

![Diagram](image-url)

*Figure 1-1 Reverse Software Engineering*
Reverse engineering is not only motivated for re-documenting the legacy software, but also to keep up the upcoming source code and its documentation consistent [16].

Our project will implement the translation from the program source code to Modeling Text Representation Language (MTRL) and subsequently to a UML description. Chapter 2 offers more details about our strategy.

1.3 A Simple Example

With having our problem described previously, this sub-section illustrates the actual workings of our project in brief. In this example, an intermediate language is generated from a sample Java source code, and a corresponding UML class diagram is drawn.

The Circle class consists of two attributes: Radius and Center and four operations: setRadius(), setCenter(), getArea() and getCircumference(). Following is the code snippet of the class Circle:

```java
public class Circle
{
    private double Radius;
    private Point Center;

    public void setCenter(Point c){ Center = c;}
    public void setRadius(double r){ Radius = r;}
    public double getArea(){ return 3.14 * Radius * Radius;}
    public double getCircumference(){ return 2 * 3.14 * Radius;}
}
```

The intermediate language comprises of the listings of the attributes and operations that were declared in the class Circle. The structure of the above class is described in MTRL below:

```xml
<object class
    name "Circle"
    id "123"
attributes(list Attributes
    (object classAttribute
        name "Radius"
        id "1234"
        returnType "double"
        access "private"
    )
    (object classAttribute
        name "Center"
        id "1235"
        returnType "Point"
        access "private"
    )
)
operations(list Operations
    (object classOperation
        name "setCenter"
```
id “12311”
argument “Point”
returnType NULL
access “public”)

(object classOperation
   name “setRadius”
   id “12312”
   argument “double”
   returnType NULL
   access “public”)

(object classOperation
   name “getArea”
   id “12323”
   argument NULL
   returnType “double”
   access “public”)

(object classOperation
   name “getCircumference”
   id “12334”
   argument NULL
   returnType “double”
   access “public”)
)

Now, the class diagram is drawn as shown below. A class diagram consists of three compartments. The class name is specified in the first compartment, whereas the attributes and operations, along with their access and the return type, take up the second and third compartment respectively.
1.4 Structure of Subsequent Chapters

The remainder of this report is structured as follows. Section 2 describes in detail about the algorithms used in the project followed by design in Section 3. Experimental results are covered in Section 4. Finally we conclude the project and anticipate the future work of this study.

2 The Method

2.1 Program Flow

Figure 2-1 shows the basic flow of our project. The MTRL (Modeling Text Representation Language) code generator takes the source code and the input grammar as its input. The MTRL code generator uses bidirectional parallel parsing technique described in Section 2.2 to efficiently parse the input Java source code. As a result, it produces its corresponding MTRL code, which is just a text representation of its UML class diagram. Subsequently, the MTRL code is parsed by the UML diagram generator using the MTRL grammar to produce the UML class diagram. The next section describes about MTRL.

<table>
<thead>
<tr>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Radius: double</td>
</tr>
<tr>
<td>- Center: Point</td>
</tr>
</tbody>
</table>

+ setCenter(c:Point): void
+ setRadius(r:double): void
+ getArea(): double
+ getCircumference(): double
In this section, we present our MTRL in detail. A simple MTRL program is given for instance.

The MTRL basically consists of objects. Each object may have a set of attributes that gives information about its structure, and a set of operations that gives information about its behavior. Initialization of the attributes is done within the definition of the object to which they belong. The attributes are initialized as follows: `attributeName attributeValue`, where `attributeName` and `attributeValue` denotes the name and the value of the attribute, respectively.

In MTRL, the attributes are also classified according to their occurrence in the object definition as obligatory or optional attributes. The attributes that must compulsorily occur in the object definition are the obligatory attributes. Examples of such attributes include “name” and “id” of an object. These attributes represent the name and the identity of an object respectively.

These attributes have to be instantiated in order to assure the uniqueness in the namespace. The optional attributes are those having optional occurrences in the object.

MTRL also provides two keywords which may lead to the instance generation of the UML things and collections. These two keywords are: `object` and `list`. The keyword `object` is

---

**Figure 2-1** Project flow for Reverse Engineering
used for creating an UML thing, such as classes and relationships; whereas the keyword list is used for creating a collection type object or UML containers, such as packages and views.

The syntax of an instance of a UML thing type is:

\[
\text{(object } \langle \text{ObjectType} \rangle \langle \text{AttributesList} \rangle)\]

where \( \langle \text{ObjectType} \rangle \) is the predefined name of MTRL objects and \( \langle \text{AttributesList} \rangle \) is a sequence of initial values of the corresponding attributes of \( \langle \text{ObjectType} \rangle \).

The syntax of the creation of a UML collection type object is:

\[
\text{(list } \langle \text{ListType} \rangle [\langle \text{objectList} \rangle])\]

where \( \langle \text{ListType} \rangle \) is the type of list and the optional \( \langle \text{objectList} \rangle \) represents a sequence of objects.

The MTRL specification shown below represents a simple abstract UML class icon divided into three compartments. The first compartment is class1, the class name, having a static attribute called attribute1 in its second compartment. The third compartment is empty as there are no operations. In addition, the text representation in MTRL is shown below. This represents an example of the input word for our bidirectional parser.

\[
\text{(object Class attributes(list Attributes operations(list Operations) name class1 id 123 documentation stringDoc abstract true )})}
\]

\[
2.1.2 \quad \text{MTRL Grammar Rules}
\]

This section presents a context free grammar to generate the tokens associated with a class. Its start symbol is the non-terminal CLASS. As a remark, the non-terminal symbols are written by convention with all capital letters, and all the rest are terminal symbols. The set of productions is:

1. \( \text{CLASS } \rightarrow \{ \text{ object Class ATTR_LIST OPER_LIST TYPE_DEF } \} \)
2. \( \text{ATTR_LIST } \rightarrow \text{ attributes } ( \text{ list Attributes ATTIBUTES } ) \)
3. \( \text{OPER_LIST } \rightarrow \text{ operations } ( \text{ list Operations OPERATIONS } ) \)
This section focuses on a parallel algorithm that describes the general bidirectional parsing strategy [2].
2.2.1 Consideration

A SIMD (simple instruction stream, multiple data stream) computer is taken as a model. Two processors, P1 and P2, are considered. Moreover, the model shares a common memory, and the processors operate simultaneously and synchronously.

2.2.2 Variables

Here is a list of variables the algorithm has utilized:

- \( w \in V_T^* \) contains the input word (stored in the common memory), where \( V_T \) is a set of terminal symbols;
- \( n = |w| \);
- \( i_1, i_2 \) are two counters used for indicating the positions of the pointers to \( w \) (stored in the common memory);
- \( \text{accept} \) is a boolean variable which takes the value true if and only if \( w \in L(G) \) (stored in the common memory);
- \( \text{Stack1} \) and \( \text{Stack2} \) are two working stacks for processor P1 and processor P2;
Output_tape1 and Output_tape2 are the output tapes of P1 and P2 for storing the partial syntactic analysis of w;
Output_tape is the output tape for storing the global syntactic analysis of w (stored in the common memory);
exit is a boolean variable which is true if and only if P1 or P2 detect the non-acceptance of w (stored in the common memory).

2.2.3 Predefined procedures

The procedures predefined in this project are:

- pop(Stack, α) – the resulting value of α will be the string of length |α| starting from the first symbol of Stack; after that, the string α is removed from Stack;
- push_first(Stack, A) – adds the symbol A to the content of Stack; A will be the new top of Stack;
- push_last(Stack, α) – adds the string α to the content of Stack, starting from the last symbol of Stack; Stack will have the same top.
- ACTION(i, a) – Takes a state i and a terminal a as arguments. There could be four possible outputs:
  a) Shift j, where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses j to represent a.
  b) Reduce A → β. The action of the parser effectively reduces β on the top of the stack to head A.
  c) Accept. The parser accepts the input and finishes parsing
  d) Error. The parser discovers an error.
- GOTO(Ii ,A) – Output is a state Ij. In other words, the operation maps a state i and a non-terminal A to state j.

2.2.4 Algorithm

Let the context free grammar G = (VN, VT, S, P) be given, where VN is the set of non-terminals, VT is the set of terminals, S is the start symbol (S ∈ VT), and P is the set of productions. The pseudo-code of the parallel bidirectional parser is given below:

begin
read(n); read(w); i1 := 1; i2 := n;
accept := false; exit := false;
repeat in parallel
  action1(P1);
  action2(P2)
until (exit1 = true && exit2 = true) || (accept1 = false || accept2 = false));
if(accept1 = true && accept2 = true) then begin
  repeat
    action3(P1, P2)
  until (exit = true);
end
if (accept = true) then begin
  write('w is accepted and has right hand syntactical analysis');
  write(Output_tape)
end
else write('w is not accepted');
end.

**Procedure action1**(P1) is in charge of parsing the input word from left-to-right by processor P1 and is given by:

```plaintext
procedure action1(P1);
begin
  accept1 := true;
  case
    let s be the current state
    if (ACTION(s, w[i1]) = reduce r1 : A → α)
      then begin
        /* reduce action */
        pop(Stack1, α);
        push_first(Stack1, A);
        pop |α| symbols off the state_stack1;
        let t1 now be on top of state_stack1;
        push_first(state_stack1, GOTO(t1, A));
        push_first(Output_tape1, r1);
      end;
    if (i1 <= i2 and ACTION(s, w[i]) = shift) then begin
      /* shift action */
      push_first(Stack1, w[i1]);
      i1 := i1 + 1;
    end
    if (ACTION(s, w[i1]) = error) then begin
      /* error action */
      accept1 := false;
      exit1 := true;
    end
    if (no more shift or reduce action could be made)
      then exit1 := true;
  end;
end;
```

**Procedure action2**(P2) is similar to **action1**(P1) and is in charge of parsing the input word from right-to-left by processor P2.

```plaintext
procedure action2(P2);
begin
  accept2 := true;
  case
    let s be the current state
    if (ACTION(s, w[i1]) = reduce r2 : B → β)
      then begin
        /* reduce action */
        pop(Stack2, β);
        push_first(Stack2, B);
        pop |β| symbols off the state_stack2;
        let t2 now be on top of state_stack2;
        push_first(state_stack2, GOTO(t2, B));
        push_first(Output_tape2, r2);
      end;
    if (i1 < i2) then begin
      /* shift action */
      push_first(Stack2, w[i2]);
      i2 := i2 - 1;
    end
  end;
end;
```
Finally, the procedure action3(P1,P2) is described in a sequential way. The input tape is now empty, i.e., w has been already read (of course, if exit has the value false). Next we read symbols from Stack2 (sent by processor P2) modifying the content of the Output_tape1 and Output_tape2 putting the results in Output_tape.

procedure action3(P1,P2);
begin
  case
    let s be the current state and a be the current input symbol
    if (ACTION(s, a) = reduce r1 : A → α)
      then begin
        /* reduce action */
        pop(Output_tape1, π₁');
        pop(Stack1, α);
        push_first(Stack1, A);
        pop |α| symbols off the state_stack1;
        let t₁ now be on top of state_stack1;
        push_first(state_stack1, GOTO(t₁, A));
        push_first(Output_tape, r₁);
        push_last(Output_tape, 1');
      end;
    if (top of Stack2 is a terminal symbol) then begin
      /* shift-terminal action */
      pop(Stack2, a), where a is a terminal symbol ;
      push_first(Stack1, a);
    end;
    if (top of Stack2 is a non-terminal symbol) then begin
      /* shift-nonterminal action */
      pop(Stack2, A);
      push_first(Stack1, A);
      pop(Output_tape2, π₂');
      push_first(Output_tape, π₂');
    end;
    if(ACTION(s, a) = error) then begin
      /* error action */
      accept := false;
      exit := true;
    end;
    if (Output_tape1 = ∅) and (Output_tape2 = ∅) and
       (Stack1 = S) and (Stack2 = ∅) then begin
      accept := true; exit := true
    end;
  end;
### 2.2.5 Example of applying the previous parallel parsing algorithm

Let $G = ((A,B,C), \{a,b,c,d,e\}, A, P)$ be a context free grammar, where the set of productions $P$ is given by:

1. $A \rightarrow a B A C b$
2. $A \rightarrow d A$
3. $A \rightarrow e$
4. $B \rightarrow b B c$
5. $B \rightarrow d$
6. $C \rightarrow c C$
7. $C \rightarrow d$

The parsing table for the above grammar is shown in Table 2-1.

<table>
<thead>
<tr>
<th>STATE</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>$$</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>acc</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>r7</td>
</tr>
<tr>
<td>11</td>
<td>s12</td>
</tr>
<tr>
<td>12</td>
<td>r1</td>
</tr>
<tr>
<td>13</td>
<td>r2</td>
</tr>
<tr>
<td>14</td>
<td>s15</td>
</tr>
<tr>
<td>15</td>
<td>r4</td>
</tr>
<tr>
<td>16</td>
<td>r6</td>
</tr>
</tbody>
</table>

**Table 2-1 Parsing table for grammar G**

Let us consider an input word, $w = adabcdedbcdb$. We will parse this word using both the sequential as well as the parallel techniques. The sequential parsing algorithm is described below. Whereas, the up-to-up bidirectional parsing algorithm described in Section 2.2.4 would be used for the parallel parsing.

Let $a$ be the first symbol of $w$;

```plaintext
while(1) { /* repeat forever */
    let $s$ be the state on top of the stack
    if ( ACTION[$s$, $a$] = shift $t$ ) {
        push $t$ onto the stack;
        let $a$ be the next input symbol;
    }
    else if ( ACTION[$s$, $a$] = reduce $A \rightarrow \beta$ ) {
        pop $|\beta|$ symbols off the stack;
        let state $t$ now be on the top of the stack;
        push GOTO[$t$, $A$] onto the stack;
        output the production $A \rightarrow \beta$;
    }
}
else if ( ACTION[s, a] = accept ) break; /* parsing is done */
else error;
}

Using the above sequential algorithm, the sequence of the stack and input contents for the input word, $w = \text{adabcdedbc}cd$, is shown in the Figure 2-3.
<table>
<thead>
<tr>
<th>STACK</th>
<th>SYMBOLS</th>
<th>INPUT</th>
<th>ACTION</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>adabdcdedbccdb$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>a</td>
<td>dabcdedbccdb$</td>
<td>shift</td>
</tr>
<tr>
<td>3</td>
<td>027</td>
<td>ad</td>
<td>abcdedbccdb$</td>
<td>reduce B → d</td>
</tr>
<tr>
<td>4</td>
<td>025</td>
<td>aB</td>
<td>abcdedbccdb$</td>
<td>shift</td>
</tr>
<tr>
<td>5</td>
<td>0252</td>
<td>aBa</td>
<td>bcdedbccdb$</td>
<td>shift</td>
</tr>
<tr>
<td>6</td>
<td>02526</td>
<td>aBab</td>
<td>cdedbccdb$</td>
<td>shift</td>
</tr>
<tr>
<td>7</td>
<td>025267</td>
<td>aBabd</td>
<td>ededbccdb$</td>
<td>reduce B → d</td>
</tr>
<tr>
<td>8</td>
<td>0252614</td>
<td>aBabB</td>
<td>cdedbccdb$</td>
<td>shift</td>
</tr>
<tr>
<td>9</td>
<td>025261415</td>
<td>aBabBc</td>
<td>dedbccdb$</td>
<td>reduce B → bBc</td>
</tr>
<tr>
<td>10</td>
<td>02525</td>
<td>aBaB</td>
<td>dedbccdb$</td>
<td>shift</td>
</tr>
<tr>
<td>11</td>
<td>025253</td>
<td>aBaBd</td>
<td>edbccdb$</td>
<td>shift</td>
</tr>
<tr>
<td>12</td>
<td>0252534</td>
<td>aBabde</td>
<td>dbccdb$</td>
<td>reduce A → e</td>
</tr>
<tr>
<td>13</td>
<td>02525313</td>
<td>aBabdA</td>
<td>dbccdb$</td>
<td>reduce A → dA</td>
</tr>
<tr>
<td>14</td>
<td>025258</td>
<td>aBaBA</td>
<td>dbccdb$</td>
<td>shift</td>
</tr>
<tr>
<td>15</td>
<td>02525810</td>
<td>aBabAd</td>
<td>bcdb$</td>
<td>reduce C → d</td>
</tr>
<tr>
<td>16</td>
<td>02525811</td>
<td>aBabAC</td>
<td>bcdb$</td>
<td>shift</td>
</tr>
<tr>
<td>17</td>
<td>0252581112</td>
<td>aBabACb</td>
<td>ccdb$</td>
<td>reduce A → aBACb</td>
</tr>
<tr>
<td>18</td>
<td>0258</td>
<td>aBA</td>
<td>ccdb$</td>
<td>shift</td>
</tr>
<tr>
<td>19</td>
<td>02589</td>
<td>aBAc</td>
<td>cd$</td>
<td>shift</td>
</tr>
<tr>
<td>20</td>
<td>025899</td>
<td>aBAcc</td>
<td>db$</td>
<td>shift</td>
</tr>
<tr>
<td>21</td>
<td>02589910</td>
<td>aBAccd</td>
<td>b$</td>
<td>reduce C → d</td>
</tr>
<tr>
<td>22</td>
<td>02589916</td>
<td>aBAccC</td>
<td>b$</td>
<td>reduce C → cC</td>
</tr>
<tr>
<td>23</td>
<td>0258916</td>
<td>aBacC</td>
<td>b$</td>
<td>reduce C → cC</td>
</tr>
<tr>
<td>24</td>
<td>025811</td>
<td>aBAC</td>
<td>b$</td>
<td>shift</td>
</tr>
<tr>
<td>25</td>
<td>02581112</td>
<td>aBACb</td>
<td>$</td>
<td>reduce A → aBACb</td>
</tr>
<tr>
<td>26</td>
<td>01</td>
<td>A</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Figure 2-3 Moves of sequential parser on the input word \( w = adabdcdedbccdb \)

Now, let us observe how the parallel parsing algorithm executes the above input word. Let us split the input word into two parts so that one processor parses the first part, and the other processor works on the second part of the input word. The parsing table for the first parser will be the same as given above (Table 2-1). As for the second parser, its table is given below:
The sequence of procedure action1(P1), which is in charge of parsing the input word from left-to-right by processor P1, is shown in Figure 2-4.

Likewise, the sequence of procedure action2(P2), which is in charge of parsing the input word from right-to-left by processor P2, is shown in the figure below:
After the parallel algorithm meets the middle of the input word, only one processor will be active to complete the remaining job. In this case, we assume that processor \( P_1 \) becomes active and reads symbols from Stack2 and outputs from Output2. As described in the algorithm, \( \text{action3}(P_1, P_2) \) will begin thereafter. The sequence of \( \text{action3}(P_1, P_2) \) is given below:

<table>
<thead>
<tr>
<th>STACK</th>
<th>SYMBOLS</th>
<th>INPUT</th>
<th>ACTION</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 2 5 2 5 3</td>
<td>aBaBd</td>
<td>eCbCb$</td>
<td>shift</td>
<td>4 5 5 5</td>
</tr>
<tr>
<td>2 0 2 5 2 5 3 4</td>
<td>aBaBde</td>
<td>CbCb$</td>
<td>reduce A \rightarrow e</td>
<td>3 4 5 5 5</td>
</tr>
<tr>
<td>3 0 2 5 2 5 3 13</td>
<td>aBaBdA</td>
<td>CbCb$</td>
<td>reduce A \rightarrow dA</td>
<td>2 3 4 5 5</td>
</tr>
<tr>
<td>4 0 2 5 2 5 8</td>
<td>aBaBA</td>
<td>CbCb$</td>
<td>shift</td>
<td>7 2 3 4 5 5</td>
</tr>
<tr>
<td>5 0 2 5 2 5 8 11</td>
<td>aBaBAC</td>
<td>bCb$</td>
<td>shift</td>
<td>7 2 3 4 5 5</td>
</tr>
<tr>
<td>6 0 2 5 2 5 8 11 12</td>
<td>aBaBACb</td>
<td>Cb$</td>
<td>reduce A \rightarrow aBACb</td>
<td>1 7 2 3 4 5 5</td>
</tr>
<tr>
<td>7 0 2 5 8</td>
<td>aBA</td>
<td>Cb$</td>
<td>shift</td>
<td>1 7 2 3 4 5 5</td>
</tr>
<tr>
<td>8 0 2 5 8 11</td>
<td>aBAC</td>
<td>b$</td>
<td>shift</td>
<td>6 6 7 1 7 2 3 4 5 5</td>
</tr>
<tr>
<td>9 0 2 5 8 11 12</td>
<td>aBACb</td>
<td>$</td>
<td>reduce A \rightarrow aBACb</td>
<td>1 6 6 7 1 7 2 3 4 5 5</td>
</tr>
<tr>
<td>10 1</td>
<td>A</td>
<td>$</td>
<td>accept</td>
<td>1 6 6 7 1 7 2 3 4 5 5</td>
</tr>
</tbody>
</table>

**Figure 2-5** Sequence of procedure \( \text{action2}(P_2) \) by processor \( P_2 \)

From the above figure, we can observe that the parallel bi-directional parser accepts the input word and produces the same output as produced by the sequential parser.
3 Design

3.1 System and Tools

To design the project we have used following system, library functions and tools:

3.1.1 System

We used the following system:

ii. Memory (RAM): 3 GB
iii. Processor: Intel® Core™ 2 Duo 2.4 GHz., L2 Cache Memory = 4 MB

3.1.2 Library Functions

The library functions used in designing our project includes:

i. Parallel Java Library [10]: Parallel Java (PJ) is an API and middleware for parallel programming in 100% Java on shared memory multiprocessor (SMP) parallel computers, cluster parallel computers, and hybrid SMP cluster parallel computers. PJ was developed by Professor Alan Kaminsky and his student Luke McOmber in the Department of Computer Science at the Rochester Institute of Technology.
ii. JGraph 5 [9]: It is the most powerful, easy-to-use, feature-rich and standards-compliant open source (BSD) graph component available for Java. This API is used for generating UML diagram.

3.1.3 Tools

The tools that assist us for developing this project are:

i. JDK 1.6.11: It is a software development environment for writing applets and application in Java.
ii. NetBeans IDE 6.8 [6]: It is a platform framework for Java desktop applications, and an integrated development environment (IDE) for developing with Java and some other programming languages, such as PHP, C++, etc. The NetBeans IDE is written in Java and runs everywhere where a JVM is installed, including Windows, Mac OS, Linux, and Solaris. A JDK is required for Java development functionality, but is not required for development in other programming languages.

3.2 Class Diagram

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system: conceptual, specification, and implementation [11]. These perspectives become evident as the diagram is created and help solidify the design. Figure 3-1 depicts the class diagram for the project. It shows the classes of the system, their interrelationships, and their
operations and attributes. Classes are depicted as boxes with three sections: the top one indicates the name of the class, the middle one lists the attributes of the class, and the third one lists the methods.

**Figure 3-1** Class diagram for the project

**PSE**: It is the core application class, implemented as Singleton. The class acts as a control board for all the provided functionality, such as loading source code, displaying the UML diagram, quitting the application, etc.
**DesignExtractor:**

The main functionality of this class is to convert a source code, for example Java source code, to its equivalent MTRL code. Given a directory that contains the source codes, it generates their corresponding MTRL codes.

**SourceParser:**

A bidirectional parser for object oriented programming language.

**UMLDrawer:**

This class effectively produces the UML model. It takes MTRL code as the input.

**MTRLParser:**

A bidirectional parser for MTRL.

**SourceGenerator:**

It generates source code in a specified object-oriented programming language, provided its corresponding MTRL code.

### 4 Experimental Results

In this section, we tried to find out how the parallel bidirectional parser executed for an MTRL code. Let us consider a simple MTRL code with 31 tokens as given below:

```
(object Class
  attributes(list Attributes
    (object ClassAttribute
      name attribute1
      id 1234
      exportControl "Protected"
    )
  )
  operations(list Operations)
    name class1
    id 123
    abstract true
  )
)
```

We pointed out the iterations of the procedures \texttt{action1(P1)} and \texttt{action2(P2)}. Because processors \texttt{P1} and \texttt{P2} worked independently, it implied that the execution was non-deterministic. We got different intermediary results, even if the right-most derivation was the same. One of the executions we obtained is the following. Using the production rules given in Section 2.1.2, the procedure \texttt{action1(P1)} had the partial syntactic analysis:

\[ \Pi_{P1} = [13; 18; 32; 28; 13; 16; 34; 24; 14; 15; 35; 19] \]
and the execution of action2(P2) was then:
\[ \Pi_{P2} = [13; 17; 31; 25; 13; 16; 34; 24; 14; 15; 35; 19] \]

Processors P1 and P2 met somewhere in the “middle” of the input word, and P1 became more active than P2. Processor P2 only sent the data (i.e., Stack2, Output_tape2) to processor P1. In the end, we deduced that the word w is accepted by the bidirectional parser, and it had the right-most syntactic analysis:
\[ \Pi_{rm} = [1; 13; 18; 32; 28; 13; 16; 34; 24; 14; 15; 35; 19; 3; 7; 2; 4; 9; 10; 13; 17; 31; 25; 13; 16; 34; 24; 14; 15; 35; 19] \]

Next, we compared the execution time of the parallel parser with a serial LR parser. We ran both implementations on the same machine with Intel® Core™ 2 Duo 2.4 GHz processor and 3 GB of main memory.

![Figure 4-1 Comparison of parallel and serial parser](image)

The above figure compares the execution time between the parallel bidirectional parser and the serial LR parser. We generated few random MTRL codes and the X-axis shows the number of tokens in those examples. We run those examples using both the parallel and the serial versions of the parser on the same machine. The Y-axis shows the execution time in number of seconds.

We can observe that for a small program (that has a less number of tokens), the parallel and the serial parser have nearly same execution time. However, we can notice that for a large program (that has a more number of tokens), the execution time for the parallel version of the parser is distinguishably lesser than that of the serial version.
5 Conclusions and Future Works

Automated software engineering has been a challenging topic. Its goal is to partially or fully automate software engineering activities, thereby significantly increasing both quality and productivity. In this project, we were successfully able to mechanize the process of generating UML class diagrams from a given Java source code. In addition, we were able to produce Java source code templates that correspond to an object model. Many modeling constructs, such as attribute and association specification, were automatically mapped to Java source code, while the bodies and additional private methods need to be added later by developers.

With current computer architectures increasingly relying upon hardware level parallelism, we developed a bidirectional parallel parser to improve the performance of our project. As a result, we were able to gain increased performance for large programs.

In the future, this project can be extended to support multiple object oriented programming languages such as C++, Python, C#, PHP, etc. In addition, with the recent inventions of Quad-core and Core i7 processors, it is also possible for the project to support more than two processors.

References


