Parallel Parsing-based Reverse Engineering

Stefan Andrei

No. 3, September 2008
Parallel Parsing-based Reverse Engineering

Ștefan Andrei
Department of Computer Science, Lamar University
Beaumont, Texas, U.S.A., sandrei@cs.lamar.edu

September 30, 2008

Abstract

Parsing descriptive programming languages, such as eXtendable Markup Language (XML) and Unified Modeling Language (UML), has been an active area of research. Lam, Ding, and Liu claim that among all important phases of XML (e.g., parsing, access, modification, and serialization), parsing is the most time-consuming one. That motivates investigation of efficient parsing techniques with applications in many computer science areas, including reverse engineering.

While there are many works on parsing XML, there is still room for research about UML parsing. The reason that makes UML parsing challenging is that UML deals with graphical representations, such as class icons, class diagrams, sequence diagrams, state diagrams, and not text representations that are input for traditional parsers. Reverse engineering is an important sub-area in software engineering and it basically means obtaining the UML specification from a source program described in an (object-oriented) programming language.

This paper describes a new application of a non-traditional parallel (e.g., bidirectional) parsing technique for a Uniform Petal Language (UPL) program. Our bidirectional parser takes as input a UPL program that is the text representation of a UML specification. The benefits of parsing a UPL program are checking the correctness of the UML specification and obtaining the productions that lead to that UPL program. Consequently, the UPL program can be generated from the associated set of productions. The set of productions associated to a correct UPL program is a very condensed way to store it. Keeping a specification as minimum as possible, but expressive, is
an important concept of reverse engineering. We have implemented a bidirectional parser for context free languages, including UPL, in Java programming language, called Parsing Uniform petAl Language (PU-PAL), and compare it with CUP, a state-of-the-art parser for Java designed by Hudson, Flannery, and Ananian. The experimental results considered several UPL programs and concluded that PUPAL performs better than CUP.

1 Introduction

Parsing programs written in a programming language has been an important research area since 70’s. Once a programming language evolves over the years in order to cover higher level programming concepts, other aspects of that programming language needed further attention, such as code optimization, memory allocation, formal code proofs, and so on. Hence, there is a belief that parsing occupies only a small percentage of the compilation process for a given program compared with other compilation phases [2]. Despite this belief, there exist many programming languages where the parsing phase represents an important and active research topic [10]. According to [7], an XML parser takes as input a raw serialized string and performs certain operations on it. First it checks the syntactic well-formedness of the XML data, making sure that the start tags have matching end tags and that there are no overlapping elements. Most parsers also implement validation against the Document Type Definition (DTD) or the XML Schema to verify that the structure and content are as specified. Finally, the parsing output provides access to the content of the XML document via programmatic APIs.

According to [12], the XML parsing is the most time-consuming phase compared with the other three XML phases, namely access, modification, and serialization. The authors of [12] indicate the XML parsing has three stages: character conversion, lexical analysis (these are usually invariant among different parsing models) and syntactic analysis (that creates data representations based on the parsing model used).

Efforts have been made for implementing parser generators for any arbitrary programming language. For instance, CUP is a parser generator for Java implemented by Scott Hudson, Frank Flannery, C. Scott Ananian at Georgia Tech in 1999 [8]. Currently, the CUP parser generator is maintained at the Technical University of Munich, Germany. The newest CUP devel-
Development version was released in June 2006 and is able to handle the CUP embedded actions [9].

The Unified Modeling Language (UML) is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems [1]. 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. 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 [6, 11, 13] at Rational Software Corporation, with contributions from other leading methodologists, software vendors, and many users. Based on extensive use of the Booch and Jacobson methods, UML is the evolution of these and other 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. Rational Software Corporation provides a professional software tool called the Rational Rose Enterprise Edition. A partial list of the features of The Rational Rose 2000e Enterprise Edition is UML modeling, multi-language development in C++ and Java, round-trip engineering, component-based development, extensibility interface, and so on.

Because the structure of the system defined by the diagram is translated by a developer into actual source code (classes), the UML tool should bridge this step by generating the source code of the classes with the methods stubbed out. Software developers can use this stub code and fill in with the actual code. This characteristic of automating the generation of source code is called forward software engineering (Figure 1). Forward engineering supported by a UML tool is normally for a specific language or a set of languages, e.g., Java, C++, and so on. The UML tools (such as Rational Rose) start with a visual presentation of a UML model and provide in the output a text representation. This text representation is called UML Petal Language (UPL). UPL is a compiled language useful for building UML semantic models. Based on these, some visual presentations may be generated.
Reverse software engineering is exactly the opposite of forward software engineering (Figure 2). 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, to estimate costs of components, and so on.

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 [15].

The development of the Extensible Markup Language (XML), the many free tools and application programmer interfaces for XML-based processing, and the inclusion of XML support in many commercial products has led to a standardization for the transport of structured data across the Web [5]. Similarly, the Resource Description Framework (RDF) and the development of publicly available RDF tools has led to RDF becoming an emerging standard for expressing simple metadata content, with RDF Schema providing a simple mechanism for defining metadata schemas or ontologies. RDF is a language based on resource-property-value triples designed for expressing statements about resources on the Web (or anything that has an associated uniform resource identifier). An RDF model consists of a set of triples, each
of which can be regarded as a simple logical proposition about a resource. To generate RDF schemas from UML class diagrams and to allow instances of these schemas (i.e., knowledge) to be expressed in RDF, two mappings were defined:

1. From UML class diagrams to RDF schemas;
2. From UML class diagrams to sets of Java classes.

The mappings from UML class diagrams to RDF schemas and sets of Java classes were implemented using two stylesheets in the XSLT language. The inputs to the stylesheets are XMI encodings of class diagrams (using XMI version 1.0 for UML1.3 - the export format supported by the tools Argo/UML 0.8 and Rational Rose 2000 with the Unisys XMI). An XSLT stylesheet is comprised of a set of templates that match nodes in the input document, represented internally as a tree, and transform them to produce an output tree, represented as text or as an HTML or XML document. For instance, Xpetal is a tool that converts models in the UPL format.

Motivation: The paper contribution is two-fold. Firstly, we define a novel context-free grammar able to generate a formal language for analyzing the UPL text representations for correctness. Secondly, once the UPL representation is correct, we generate the set of grammar productions able to describe the corresponding UML model (this relates to the last two layers of Figure 2). Moreover, we implemented a Java programming language tool for doing the two phases of analysis and generation. The experimental results demonstrate that our technique is efficient compared with other state-of-the-art parsers.

The structure of this paper: Section 2 presents some details about UPL. Section 3 describes an efficient bidirectional parser for UPL and a context free grammar that generates a non-trivial subset of UPL. Experimental results (Section 4) demonstrate the applicability of our parsing method for a simple UPL example, as well as a comparative evaluation of our tool on several other examples. Conclusion ends the paper.

2 The Uniform Petal Language

There are only few references about the UPL. This section gives some insights about UPL as a text representation of UML as well as an example of a simple
UPL program. UPL has the main goal to generate instances of some classes of hierarchy (called UMLHierarchy) which represent the basement of the entire language. There are two types of abstractions within this hierarchy:

- abstractions of the **UML things**, such as classes (with their attributes and operations), relationship, notes, and so on;
- abstractions of the **UML containers** (or collections), such as packages, views, diagrams, but also some sets which even they are not UML collections, they are used by certain UML things to store possible specific data (i.e., sets of attributes and operations of classes).

In UPL, each object may have a set of attributes which gives its **structure** and **behavioral** as well. The attributes may be initialized within the definition of the object to which they belong, according to the following syntax: `attributeName attributeValue`, where `attributeName` is the name of the attribute and `attributeValue` is the value of the attribute.

The attributes may be also classified after their occurrence in the object definition: **obligatory** or **optional** attributes. The **obligatory attributes** are those having compulsory occurrence in the object definition. Such attributes are “name” and “quid” representing the name and the ID of an object. 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. UPL provides two keywords which may lead to the instance generation of the UML things and collections, corresponding to the two types of abstractions. These two keywords are: `object` and `list`; `object` is used for creating an UML thing, and `list` is used for creating a collection type object. The **syntax of an instance** of a **UML thing type** is:

```
(object <ObjectType> <AttributesList>)
```

where `<ObjectType>` is the predefined name of UPL objects and `<AttributesList>` is a sequence of initial values of the corresponding attributes of `<ObjectType>`. The **syntax of the creation** of a **UML collection type object** is:

```
(list <ListType> [objectList])
```

where `<ListType>` is the type of list and the optional `<objectList>` represents a sequence of objects. There are two types of collections: **homogeneous** (involving elements of the same type) and **heterogenous** collections (involving
elements of different type). The homogeneous collection is used for creating sets (which are not UML collections) useful by UML things to store their specific data (i.e., the sets of attributes and operations of a class or the sets of an operation parameters). The heterogeneous collections are used to create (add) the contents (at the old contents) of an UPL collection object (e.g., the UML thing list contained by a view, a package, or a diagram, is a heterogenous collection).

For instance, the below UPL specification represents a simple abstract UML class icon called class1, having a static attribute called attribute1. The below text representation is an input word for our bidirectional parser.

\[
(\text{object Class}
\text{  }\text{attributes ( list Attributes }
\text{  ( object ClassAttribute }
\text{  name attribute1 }
\text{  quid idAttribute1 }
\text{  documentation stringDoc }
\text{  exportControl "Protected" }
\text{  type stringType }
\text{  initv stringInitv }
\text{  static false }
\text{  derived true }
\text{  )}
\text{  )}
\text{  operations ( list Operations )}
\text{  name class1 }
\text{  quid idClass1 }
\text{  documentation stringDoc }
\text{  abstract true }
\text{  )}
\]

3 An efficient bidirectional parser for UPL

We suppose the reader familiar with the traditional notions as context free grammars, words ([2,14]). Let \( G = (V_N, V_T, S, P) \) be a context free grammar, where \( V_N \) is the set of non-terminals, \( V_T \) is the set of terminals, \( S \) is the start symbol \( S \in V_N \), and \( P \) is the set of productions. Let \( V = V_N \cup V_T \) be the
set of all symbols and $V'$ be the set $V_N \times N \times N$, where $N$ is the set of natural integers. Let $h : V \cup V' \rightarrow V$ be a function given by $h(X) = X$, $\forall X \in V$ and $h(X_{b,e}) = X$, $\forall X_{b,e} \in V'$. This function can be easily extended to an homomorphism $h : (V \cup V')^* \rightarrow V^*$ such as $h(\lambda) = \lambda$ and $h(X_1X_2...X_n) = h(X_1) \cdot h(X_2) \cdot ... \cdot h(X_n)$, $\forall X_1, X_2, ..., X_n \in V \cup V'$, $\forall n \geq 2$ (· being the words’ catenation operation and $\lambda$ being the empty word).

Our parsing technique is different than the traditional left-to-right parsing in the sense that it does independent parsing from both sides toward the “middle” of the word. Once all the letters of the input word have been read, the two distinct sentencial forms are catenated and the parsing continues in the traditional way from left-to-right. This strategy is called bidirectional ascendant parsing and has been briefly described in Figure 3 below and detailed in [4].

**Figure 3.** The bidirectional ascendant parsing

The “derivation forests” $ST_1$ and $ST_2$ will be parsed in parallel, followed by the subtree $ST_3$ parsed sequentially. Our model is a MIMD (Multiple Instruction stream and Multiple Data stream) computer [3]. We consider two processors $P_1$ and $P_2$ which operate asynchronously and share a common memory. Given a word $w$, we denote by $|w|$ its length. Our algorithm uses the following variables:

- $w \in V_T^*$ is the input word, $n = |w|$;
- i1 and i2 are two counters for the current positions in $w$;
- accept is a boolean variable which takes the true value if and only if $w \in L(G)$;
- Stack1 and Stack2 are two working stacks for $P_1$ and $P_2$;
- **Output_tape1** and **Output_tape2** are the output tapes of P1 and P2 for storing the partial syntactic analysis;
- **Output_tape** is the output tape for storing the global syntactic analysis;
- **exit** is a boolean variable which is true if and only if P1 or P2 detect the non-acceptance of w.

Variables w, i1, i2, accept, Output_tape, and exit are stored in the shared memory. We use some predefined procedures, such as:
- **pop(Stack, α)** - the 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)** - add to the content of Stack the symbol A; A will be the new top of Stack;
- **push_last(Stack, α)** - add to the content of Stack, starting from the last symbol of Stack, the string α; Stack will have the same top.

The main method of our bidirectional ascendat parsing algorithm is described below.

```plaintext
begin
read(n); read(w); i1:=1; i2:=n;
accept:=false; exit:=false;
repeat in parallel
  action1(P1); action2(P2)
until (i1>=i2) or (exit=true);
if (exit = true) then accept := false
else
  repeat action3(P1, P2) until (exit = true);
  if (accept = true) then
    write('w is accepted');
    write('w has right analysis', Output_tape);
  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
  case
  if (∃ r1 = A→ h(α) ∈ P, α is in Stack1
```
starting from the top) then begin
  /* reduce action */
  let \( \alpha := u_1 C_{b,e} \ldots u_m D_{t',e'} \alpha', \)
  where \( u_1, \ldots, u_m \in V_T^*, \alpha' \in V^* - V'^*; \)
  \( e'' := |Output_tape1| + 1; \)
  if \( (\alpha \cap V' - V = \emptyset) \) then \( b'' := e'' \)
  else \( b'' := \min\{e'', b\}; \)
  pop(Stack1, \( \alpha \));
  push_first(Stack1, A_{b''});
  push_first(Output_tape1, r1);
end;
if \( (i_1 \leq i_2) \) then begin
  /* shift action */
  push_first(Stack1, w[i_1]);
i_1 := i_1 + 1;
end
otherwise: begin
  /* backtrack is needed; */
  if (all backtrack steps are over) and
  (still no reduce or shift action
  could be made)
  then exit := 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.

procedure action2(P2);
begin
  case
  if \( (\exists r_2 = A \rightarrow h(\beta) \in P, \beta \text{ is in Stack2} \)
  starting from the top) then begin
    /* reduce action */
    let \( \beta := u_1 C_{b,e} \ldots u_m D_{t',e'} \beta', \)
    where \( u_1, \ldots, u_m \in V_T^*, \beta' \in V^* - V'^*; \)
    \( e'' := |Output_tape2| + 1; \)
    if \( (\beta \cap V' - V = \emptyset) \) then \( b'' := e'' \)
    else \( b'' := \min\{e'', b\}; \)
    pop(Stack2, \( \beta \));
  end
push_first(Stack2, B_{b',e'});
push_first(Output_tape2, r2);
end;
if (i1 < i2) then begin
  /* shift action */
  push_first(Stack2, w[i2]);
  i2 := i2 - 1;
end
otherwise: begin
  /* backtrack is needed; */
  if (all backtrack steps are over)
    and (still no reduce or shift action could be made) then exit := true;
end
end;

Finally, we describe the procedure action3(P1, P2). The input tape is now empty, i.e., w has been already read (of course, if exit has the value false), but we read symbols from Stack2 (send 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
    if (∃ r1 = A → h(α) ∈ P, α is in Stack1 starting from the top) then begin
      /* reduce action */
      let α = u_1 C_{b, e} ... α' D_{b', e'} α'',
      where u ∈ V_T^*, α'' ∈ V^*, α' ∈ (V ∪ V')^*;
      let π_1' from Output_tape1 such that
      |π_1'| = e' - b + 1;
      pop(Output_tape1, π_1');
      pop(Stack1, α);
      push(Stack2, A);
      push_first(Output_tape, r1);
      push_last(Output_tape, π_1');
    end;
    if (top of Stack2 is a terminal symbol) then begin
      /* shift-terminal action */

pop(Stack2,a), where a ∈ V;
push_first(Stack1,a);
end;
if (top of Stack2 is from V') then begin
    /*shift-nonterminal action*/
    pop(Stack2,A_{b,e});
push_first(Stack1,A);
    let π′_2 from Output_tape2 such that
    |π′_2| = e - b + 1;
pop(Output_tape2,π′_2);
push_first(Output_tape,π′_2);
end;
if (Output_tape1=∅) and (Output_tape2=∅) and (Stack1=S) and (Stack2=∅)
then begin
    accept:=true; exit:=true
end;
otherwise: begin
    /* backtrack is needed; */
    if (all backtrack steps are over) and
    (still no reduce or shift action could be made)
then exit := true
end;
end;

The correctness, finiteness, and complexity of the above algorithm was
detailed in [4]. Deterministic linear parallel algorithms (as particular cases of
the general ascendant bidirectional parsing algorithm) for solving the mem-
bership problem can be derived for some “combinations” of subclasses of
context free languages. The deterministic ascendant bidirectional parser has
the same model as the general one. The only difference is the uniqueness of
choosing the production r from the set of the given productions of the input
grammar (i.e., no backtracking step is needed).

Let G be a context free grammar and k be a natural number. The com-
plete definition of LR(k) grammars is given in [2]. Briefly, G is LR(k) if
k symbols from the input suffice to decide uniquely which production of G
applies in the bottom-up strategy. Given w = X_1 X_2 ... X_n a word over
V, we denote by ˜w the word X_n X_{n-1} ... X_1. Given G = (V_N, V_T, S, P)
a context free grammar, we denote by ˜G the grammar (V_N, V_T, S, ˜P), where
\( \bar{P} = \{ X \rightarrow \bar{\alpha} \mid X \rightarrow \alpha \in P \} \). We say that \( G \) is \( RL(k) \) if \( \bar{G} \) is a \( LR(k) \) grammar. A language \( L \) is \( RL(k) \) if there exists a \( RL(k) \) grammar which generates \( L \). Let \( k_1, k_2 \in \mathbb{N} \). We say that \( G \) is a \( LR(k_1) - RL(k_2) \) grammar if and only if \( G \) is both a \( LR(k_1) \) and \( RL(k_2) \) grammar. A language \( L \) is called \( LR(k_1) - RL(k_2) \) if there exists \( G \) a \( LR(k_1) - RL(k_2) \) grammar for which \( L = L(G) \).

The linear complexity of our parallel algorithm was proved in [4]. Let us denote with \( T_1(n) \), \( T_2(n) \) and \( T_3(n) \) the running time of the sequential procedures \( \text{action1}(P_1) \), \( \text{action2}(P_2) \) and \( \text{action3}(P_1,P_2) \), where \( n \) is the length of the input word. Supposing that the routing time is zero, the parallel running time \( t(n) \) satisfies the relations: \( \min\{T_1(n),T_2(n)\} + T_3(n) \leq t(n) \leq \max\{T_1(n),T_2(n)\} + T_3(n) \), and \( t(n) \in \mathcal{O}(n) \).

The context free grammar for the whole UPL is large. For the sake of presentation, we present a \( LR(1) - RL(1) \) grammar able to describe the syntax of a non-trivial UPL subclass. The grammar has 44 productions (the original \( LR(1) - RL(1) \) grammar for describing the complete UML semantic models has 96 productions). Its start symbol is the nonterminal \( \text{CLASS} \), the nonterminal symbols are written by convention with all capital letters, and the rest are terminal symbols. The set of productions is:

1. \( \text{CLASS} \rightarrow (\ \text{object Class ATTR_ATOMS ATTR_OP ATTR_NAME ATTR_QUALIFIER ATTR_DOC ATTR_ABS} ) \)
2. \( \text{ATTR_ATOMS} \rightarrow \text{attributes} (\ \text{list ATTR} ) \)
3. \( \text{ATTR} \rightarrow \text{Attributes ATTR_ATOMS} )\)
4. \( \text{ATTR} \rightarrow \text{Attributes} )\)
5. \( \text{ATTR_ATOMS} \rightarrow \text{ATTR} \)
6. \( \text{ATTR_ATOMS} \rightarrow \text{ATTR} \)
7. \( \text{ATTR} \rightarrow (\ \text{object ClassAttribute ATTR_NAME ATTR_QUALIFIER ATTR_DOC ATTR_EXP_CTRL ATTR_TYPE ATTR_INITV ATTR_STATIC ATTR_DERIV} ) \)
8. \( \text{ATTR_TYPE} \rightarrow \text{type stringType} \)
9. \( \text{ATTR_TYPE} \rightarrow \lambda \)
10. \( \text{ATTR_INITV} \rightarrow \text{initv stringInitv} \)
11. \( \text{ATTR_INITV} \rightarrow \lambda \)
12. \( \text{ATTR_STATIC} \rightarrow \text{static true} \)
13. \( \text{ATTR_STATIC} \rightarrow \text{static false} \)
14. \( \text{ATTR_STATIC} \rightarrow \lambda \)
15. \( \text{ATTR_DERIV} \rightarrow \text{derived true} \)
Using a JAVA implementation for our bidirectional parsing algorithm, we determine the viable prefix automaton for grammar $G$ which has 107 states corresponding to $LR(1)$ items with no conflicts (reduce-reduce, reduce-shift). Hence $G$ is a $LR(1)$ grammar. Moreover, we construct the viable prefix automaton for grammar $\tilde{G}$ which has 103 states corresponding to $RL(1)$ items.
with no conflicts (reduce-reduce, reduce-shift), so $\tilde{G}$ is a $RL(1)$ grammar. It follows that $G$ is a $LR(1) − RL(1)$ grammar.

4 Experimental results

In the following, we present the characteristics of the Java programming language implementation for our approach. We called the tool PUPAL (it stands for Parsing Uniform PetAl Language).

The first part of this section demonstrate how our algorithm works for the UPL example from Section 2. We shall simulate a possible execution of the bidirectional parsing algorithm for this example (processors P1 and P2 work asynchronously). We have $n = |w|$ (for the previous input word, $n = 43$) and $i_1$ and $i_2$ two pointers between 1 and 43 (notations are like in Section 3). We shall point out the iterations of the procedures action(P1) and action(P2). According to the bidirectional parsing algorithm from Section 3, the execution of these procedures will finish when $i_1 > i_2$.

Because processors P1 and P2 work asynchronously, then it implies that the execution is non-deterministic. It could happen to get different intermediary results, even if the right-most derivation is the same. So, one possible execution of our Java program is the following. The procedure action1(P1) could imply the partial syntactic analysis:

$$\pi_{P1} = [1, 36, 39, 33, 32, 17, 19, 15, 13, 10, 8];$$

and the execution of action2(P2) is then:

$$\pi_{P2} = [32, 33, 39, 42, 34];$$

Processors P1 and P2 meet somewhere in the “middle” of the input word, and P1 will be more active than P2. Processor P2 only sends the data (i.e., Stack2, Output_tape2) to the internal memory of P1. In fact, by executing the steps from procedure action(P1,P2) (from Section 3), we deduce that the word $w$ is accepted by the bidirectional ascendant parser and it has the right-most syntactic analysis:

$$\pi_{rm} = [1, 36, 39, 33, 32, 17, 19, 2, 3, 6, 7, 15, 13, 10, 8, 34, 42, 39, 33, 32];$$

The second part of this section shows a comparison between our tool and CUP [9]. We ran both implementations of PUPAL and CUP on the same machine, a Pentium IV computer system, with 3.2GHz using 2GB of main memory. Table 1 refers to six random generated UPL programs (e.g., example1 is the UPL program listed in Section 2, and the other five
examples correspond to much larger, but similar UML specifications). The third column of Table 1 is the number of tokens that UPL text representation has. The fourth and last columns show the execution time expressed in seconds of both CUP and PUPAL.

<table>
<thead>
<tr>
<th>The UPL program</th>
<th>#tokens</th>
<th>CUP sec.</th>
<th>PUPAL sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. example1</td>
<td>42</td>
<td>0.05</td>
<td>0.03</td>
</tr>
<tr>
<td>2. example2</td>
<td>163</td>
<td>0.11</td>
<td>0.07</td>
</tr>
<tr>
<td>3. example3</td>
<td>347</td>
<td>0.26</td>
<td>0.19</td>
</tr>
<tr>
<td>4. example4</td>
<td>480</td>
<td>0.36</td>
<td>0.27</td>
</tr>
<tr>
<td>5. example5</td>
<td>1206</td>
<td>1.12</td>
<td>0.84</td>
</tr>
<tr>
<td>6. example6</td>
<td>4078</td>
<td>3.88</td>
<td>2.80</td>
</tr>
</tbody>
</table>

Table 1. Comparison between CUP and PUPAL

Analyzing Table 1, we conclude that for all the considered UPL programs, PUPAL is faster than CUP by about 30%.

5 Conclusion

This paper highlights the benefits of parsing a UPL program, namely checking the correctness of the UML specification and obtaining the productions that lead to that UPL program. Our tool, PUPAL, is a Java implementation of a non-traditional parsing algorithm that takes benefits of parallel parsing strategy. Experimental results demonstrate that PUPAL is a promising and better parser, when compared with CUP, a state-of-the-art parser. Likewise CUP, PUPAL is able to parse other programming languages, not just UPL.

References


