Automated Translation of Java Control Flow into Promela

Jirapat Lueangviriyayarn, Wiwat Vatanawood

Abstract—**While software testing concerns the efficient test case design and test procedure to unveil the software defects, it is difficult to conduct the testing of the complicated and concurrent software. The formal verification of the software source code is considered as the complementary to the conventional testing. The verification model is expected and it is possibly extracted from the source code using automated scheme. Several researches have already provided the appropriate high level guidelines to translate Java into formal verification model. In this paper, we propose a development of an automated translation tool of Java source code into Proemial. The Java control flows and basic arithmetic and logical operators are focused due to the specific data types and control flows in Promela. The ANTLR tool is exploited to build our Lexer/Parser. We implement the Promela code generator based on the existing high level Java to Promela mapping ideas in terms of #define macro statements.**

Index Terms—**Translation, ANTLR, Promela**

I. INTRODUCTION

EXTRACTION of verification model from the programming language source code and subsequently followed by having it verified, are the complementary to the software testing. While the software testing concerns the efficient test case design and conducting the test procedures, the model checking of the source code would be possibly another approach to ensure the liveness and correctness of the complicated and concurrent software [1]. However, the verification model extraction task is still difficult and the formal methods related backgrounds are needed.

Currently, Java programming language is the essential and powerful in almost categories of the object-oriented software applications. Although, several researches are proposed in order to translate Java source code into verification model written in Promela [2].

In this paper, we propose an alternative scheme of automated translation of Java control flow into Promela. The input Java source code is syntactically analyzed and parsed using ANTLR tool. Then, we develop the translating tool to deal with the ANTLR's tokens which representing the structure of the original Java source code. Our translating tool generates the corresponding Promela using high level ideas for translating Java to Promela mentioned in [2].

Jirapat Lueangviriyayarn is a graduate student of Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University. Her research interest is Software Engineering (e-mail: Jirapat.Le@Student.chula.ac.th).

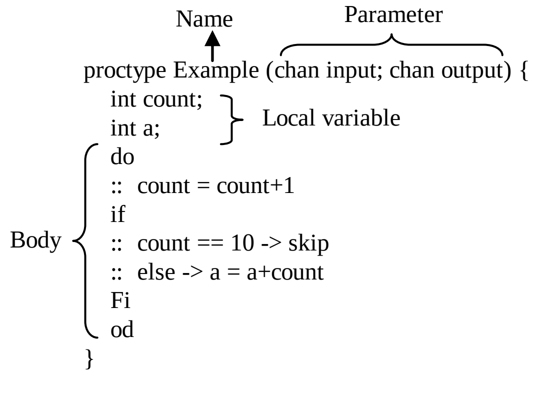
Wiwat Vatanawood is currently an Associate Professor of Computer Engineering, Faculty of Engineering, Chulalongkorn University. His research interests include Formal Specification, Formal Verification, Software Architecture. (e-mail: wiwat@chula.ac.th).

This paper is organized as follows. Section 1 is the introduction and the backgrounds are reviewed in section 2. Our scheme of automated translation of Java control flow into Promela is described in section 3. Section 4 is our conclusion.

II. BACKGROUND

A. Promela [3]

Promela is one of the well-known verification modeling language which is used to represent the properties of a formal system in SPIN. The SPIN [4] is a verification system that preferably supports the design and verification of the concurrent systems. Promela is C-like language so that it is common to most of the developers. However, the Promela is still difficult and the backgrounds in mathematical logic are needed. Typically, a Promela process is simply written as shown in Fig. 1. A process is declared by the word "prototype" following with the process name. The parameter lists would be defined along with the local variables and the process body.



```
proctype Example (chan input; chan output) {
   int count;
   int a;
   do
   ::  count = count+1
   if
   ::  count == 10 -> skip
   ::  else -> a = a+count
   Fi
   od
}
```

Fig. 1. A simple Promela process declaration

Promela is good in the definition of process and message interactions so that its data types are limited to the primitive ones, such as integer, boolean, bit, byte, unsigned, etc. While, the primitive operators are concerned on basic arithmetic operators, logical operators, bit manipulation operators, conditional comparison operators. Fig. 2. shows the data types supported by Promela and Fig. 3. shows the sample of basic control flows in Promela.

Data Types	Values	Size (bits)
bit, bool	0, 1, false, true	1
byte	0 ... 255	8
short	-32768 ... 32767	16
int	-231 ... 231-1	32
unsigned	0 ... 2n-1	≤32

Fig. 2. The data types supported by Promela [3]

Basic Control Flow	Promela Statements

atomic Sequence	atomic{
	statement_1;
	statement_2;
	}

if statement	if
	:: (a != b) -> statement_1;
	:: (a == b) -> statement_2
	fi

do-loop	do :: count = count +1 :: a = b+2 :: (count == 0) -> break od

for-loop	for(int i=0; i<5; i++){
	statements;
	}

Fig. 3. A sample of basic control flows in Promela [5]

B. *ANTLR [6]*

ANTLR (ANother Tool for Language Recognition) is a parser generator. It is widely used to build a Lexer/Parser of any language. In this paper, ANTLR needs the Java grammar rules to build a Lexer/Parser of Java source code. The Lexer performs the lexical analysis to tokenize the Java source code into a set of valid tokens. Then, the Parser does the walkthrough all of these valid Java tokens and recognizes the valid Java statements. The ANTLR needs the grammar rules written in the notations defined in [7]. In this front-end part, we use ANTLR as our tool to build Java Lexer/Parser and we also modify the parser to capture the Java tokens and rearrange the sequences of the tokens to ease our translation scheme in our back-end part.

III. Our Automated Translation Scheme

In our automated translation scheme, the Java source code is simply prepared as our input source file and the programmer would only expect the resulting corresponding Promela code in return. In this paper, the resulting Promela code is capable of representing the Object and Class, Methods, Basic mathematical operators, Methods Call, If statements and For statements, Thread and Synchronized methods. Fig. 4. shows the block diagram of our automated translation scheme.

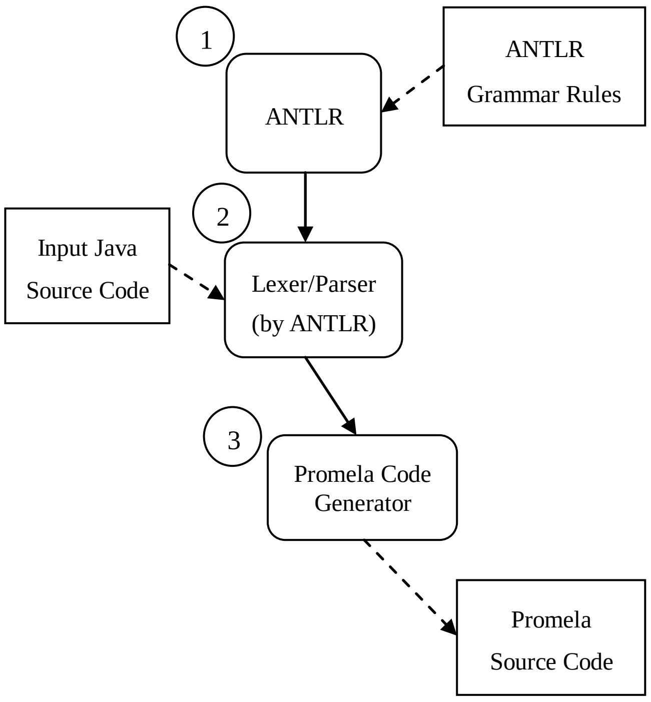


Fig. 4. Our automated translation scheme

A. *Using ANTLR to build Java Lexer/Parser*

The ANTLR is known as the universal language compiler construction tool set. In our approach, it takes the grammar rules written in EBNF that specify Java language and generates the Java language Lexer/Parser component. We can easily add some necessary actions to handle the tokens of Java to help finally generate the target Promela source code.

Our grammar rules of Java are defined based on original Java grammar rules in [8]. We embedded our additional actions into the Lexer/Parser in order to rearrange each token found in the input Java source code. The parser rules define Class, Global variable, Method, Local variable, Statement, Method invoking, If and For statement respectively.

B. *Lexer/Parser*

The Lexer/Parser is built by ANTLR according to the appropriate Java grammar rules mentioned earlier. The Lexer/Parser provides us the syntax checking tool to the input Java source code from the programmer. It is an efficient, flexible and simple way to deal with the front-end part of our automated translation tool. We are capable of supporting Java-like language, such as C# by having another C# grammar rules instead.

A Sample of the input Java source code is shown in Fig. 5. It would be analyzed by this Lexer/Parser so that the tokens would be classified and conceptually visualized as a syntax tree, shown in Fig. 6.

```
class Test {
    public int x;
    public void add2x (int d) {
        x = x+d;
    }
}
```

Fig. 5. A sample of the input Java source code

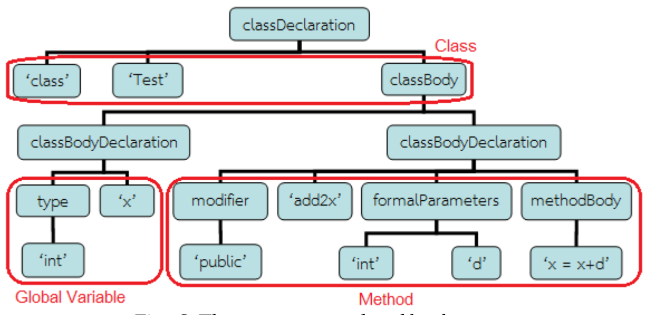


Fig. 6. The syntax tree analyzed by the parser

In our approach, the additional Java-style actions are embedded and attached to each parser rule in order to capture the tokens found at the leaves of the syntax tree, store and pass them to our next component called "Promela Code Generator." Fig. 7. shows a sample of our additional actions embedded in the ANTLR's grammar rule. The action stores all of the classes name found into a specific variable named "className" and all of the inherited classes are also stored into a variable named "classExtend". Then, both of the variables are passed to our further Promela Code Generator.

```
classDeclaration
    :   'class' n=IDENT {List className;
            if(memory.containsKey("className")){
                className = memory.get("className");
                className.add($n.text);
                memory.put("className", className);
            }else{
                className = new ArrayList();
                className.add($n.text);
                memory.put("className", className);
            }}
    typeParameters?
        ('extends' t=IDENT {List classExtend;
            if(memory.containsKey("classExtend")){
                classExtend = memory.get("classExtend");
                classExtend.add($t.text);
                memory.put("classExtend", classExtend);
            }else{
                classExtend = new ArrayList();
                classExtend.add($t.text);
                memory.put("classExtend", classExtend);
            }})?
        ('implements' typeList)?
        classBody
    ;
```

Fig. 7. Our sample action embedded in ANTLR's grammar rule

C. *Promela Code Generator*

We develop this Promela code generator according to the high level ideas for translating Java to Promela code, shown in [2]. The following Java syntaxes are considered: the Object and Class, Methods, Basic mathematical operators, Methods Call, If and For statements, Thread and Synchronized methods. However, the data types and operators of the Promela are limited to primitive ones, such as integer, boolean, bit, byte, unsigned, etc. The primitive operators are basic arithmetic operators, logical operators, bit manipulation operators, conditional comparison operators.

This Promela code generator is designed to automatically generate the final Promela code by mapping the Java statement into Promela in terms of "#define" macro statements. The difficult issues are the consistency and continuity of the resulting Promela code to represent the object instantiation and object inheritance in Java. Moreover, the Java multithreading methods using "Synchronized" and their concurrent objects are represented as well. The implementation of the guided mapping table is demonstrated in Fig. 8. and Fig. 9.

```
class X {
    public int x;
    public X() { x=0; }
    public void add2x (int d) { x = x+d;
    System.out.println("x = "+x);}
}

class XY extends X {
    public int y;
    public XY() { y = 0; }
    public void add2y (int d) { y=y+d;
        System.out.println("y = "+y);}
    public void add(int dx, int dy) {
        int old_x = x;
        int old_y = y;
        add2x(dx);
        add2y(dy);
    }
}

class Adder extends Thread {
    private XY xy;
    public Adder(XY xy) { this.xy = xy; }
    public void run() { xy.add(4,4); }
}

class Main {
    public static void main(String[] args) {
        XY xy = new XY();
        Adder adder1 = new Adder(xy);
        Adder adder2 = new Adder(xy);
        adder1.start();
        adder2.start();
    }
}
```

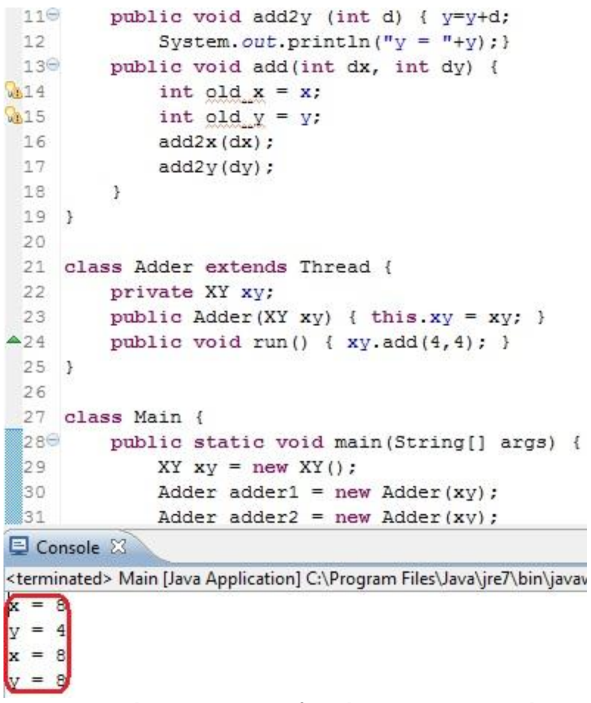
Fig. 8. Our sample of Java multithreading source code

Fig. 8. shows two Java threads named "adder1" and "adder2" and the Fig. 9. shows the resulting Promela code which correctly performs the concurrent threads of adder1 and adder2.

The instrument tests are conducted to assure the consistent behaviors between the Java version and Promela version of the codes. As shown in Fig. 10. and Fig. 11., the print out of Java code is consistent to the print out of Promela code.

```
#define ClassName mtype
#define Index byte
#define undefined 0
#define MAX 5
#define null -1
#define this _pid
ClassName = {X,XY,Adder,Main}
typedef ObjRef{ClassName class; Index index};
typedef X_Class{int x; };
X_Class X_Obj[MAX];
Index X_Next = 0;
#define X_get_x(obj)
    (obj.class == X -> X_Obj[obj.index].x :
    (obj.class == XY -> XY_Obj[obj.index].x : undefined))
#define X_set_x(obj,value)
    if :: obj.class == X -> X_Obj[obj.index].x = value
    :: obj.class == XY -> XY_Obj[obj.index].x = value fi
#define X_constr(obj)
    ObjRef obj; obj.class = X;
    atomic{obj.index = X_Next; X_Next++};
    X_set_x(obj,0);
#define X_add2x(obj,d)
    X_set_x(obj,X_get_x(obj)+d);
typedef XY_Class{int y; int x; };
XY_Class XY_Obj[MAX];
Index XY_Next = 0;
#define XY_get_y(obj)
    XY_Obj[obj.index].y
#define XY_set_y(obj,value)
    XY_Obj[obj.index].y = value
#define XY_constr(obj)
    ObjRef obj; obj.class = XY;
    atomic{obj.index = XY_Next; XY_Next++};
    X_set_x(obj,0); XY_set_y(obj,0);
#define XY_add2y(obj,d)
    XY_set_y(obj,XY_get_y(obj)+d);
#define XY_add(obj,dx,dy)
    old_x = X_get_x(obj); old_y = XY_get_y(obj);
    X_add2x(obj,dx); XY_add2y(obj,dy);
typedef Adder_Class{ObjRef xy};
Adder_Class Adder_Obj[MAX];
Index Adder_Next = 0;
#define Adder_get_xy(obj)
    Adder_Obj[obj.index].xy
#define Adder_set_xy(obj,value)
    Adder_Obj[obj.index].xy.class = value.class;
    Adder_Obj[obj.index].xy.index = value.index;
#define Adder_constr(obj,xy)
    ObjRef obj; obj.class = Adder;
    atomic{obj.index = Adder_Next; Adder_Next++};
    Adder_set_xy(obj,Adder_get_xy(obj));
proctype Adder_Thread(ObjRef obj){
    int old_x; int old_y; XY_add(Adder_get_xy(obj),4,4);
}
init{int old_x; int old_y; XY_constr(xy);
    Adder_constr(adder1,xy); Adder_constr(adder2,xy);
    run Adder_Thread(adder1); run Adder_Thread(adder2);
}
```

Fig. 9. Our resulting Promela code representing the Java threads



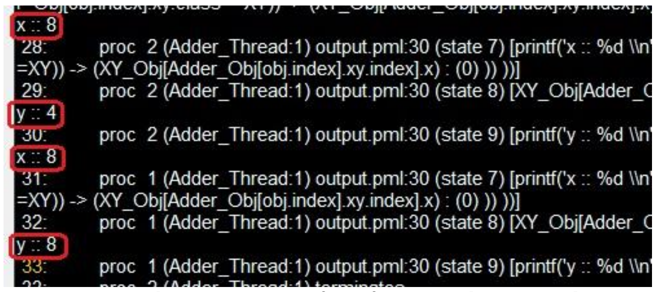
```
    public void add2y (int d) { y=y+d;
        System.out.println("y = "+y);}
    public void add(int dx, int dy) {
        int old_x = x;
        int old_y = y;
        add2x(dx);
        add2y(dy);
    }
}

class Adder extends Thread {
    private XY xy;
    public Adder(XY xy) { this.xy = xy; }
    public void run() { xy.add(4,4); }
}

class Main {
    public static void main(String[] args) {
        XY xy = new XY();
        Adder adder1 = new Adder(xy);
        Adder adder2 = new Adder(xy);
```

```
<terminated> Main [Java Application] C:\Program Files\Java\jre7\bin\javaw
x = 8
y = 4
x = 8
y = 8
```

Fig. 10. The instrument test of println() in Java source code



```
x :: 8
 28:     proc  2 (Adder_Thread:1) output.pml:30 (state 7) [printf('x :: %d \\n'
=XY)) -> (XY_Obj[Adder_Obj[obj.index].xy.index].x) : (0) )) ))]
 29:     proc  2 (Adder_Thread:1) output.pml:30 (state 8) [XY_Obj[Adder_C
y :: 4
 30:     proc  2 (Adder_Thread:1) output.pml:30 (state 9) [printf('y :: %d \\n'
x :: 8
 31:     proc  1 (Adder_Thread:1) output.pml:30 (state 7) [printf('x :: %d \\n'
=XY)) -> (XY_Obj[Adder_Obj[obj.index].xy.index].x) : (0) )) ))]
 32:     proc  1 (Adder_Thread:1) output.pml:30 (state 8) [XY_Obj[Adder_C
y :: 8
 33:     proc  1 (Adder_Thread:1) output.pml:30 (state 9) [printf('y :: %d \\n'
```

Fig. 11. The instrument test of printf() in Promela source code

IV. CONCLUSION

In this paper, we propose a development of an automated translation tool of Java source code into Promela code. Our scheme includes the front-end part called "Lexer/Parser" which is built using ANTLR tool. We modified an original Java ANTLR's grammar rules from [8] to capture and rearrange the Java tokens using our embedded actions attached to each particular parser rule. Our back-end part called "Promela Code Generator" which is designed and developed to generate the resulting Promela according to the Java to Promela mapping table guided by [2]. We focus only on the Java basic control flows which means the primitive data types and operators are only supported. Moreover, the Java multithreading method is also supported. The Promela code is generated in terms of #define macro statements and the object instantiation and inheritance are also assured. The instrument tests are conducted to assure the consistent behaviors of both source codes.

REFERENCES

[1] Christel Baier and Joost-Pieter Katoen, "Principles of Model Checking", The MIT Press Cambridge, Massachusetts London, England, 2008.
[2] Klaus Havelund and Thomas Pressburger, "Translating Java to Spin A Step Towards the JavaProver", NASA Ames Research Center, Recom Technologies, Moffett Field, California, USA. 1998.
[3] Gerard J. Holzmann, "Principles of the Spin Model Checker", Springer-Verlag London Limited, 2008.
[4] Ke Jiang, "Model Checking C Programs by Translating C to Promela", Institutionen för informationsteknologi, Department of Information Technology. September 2009.
[5] Bernhard Beckert, "Formal Specification and Verification : PROMELA", Wolfgang Ahrendt and Reiner H¨ahnle at Chalmers University, G¨oteborg.
[6] Terrence Parr. "ANTLR" [Online]. Available: http://www.antlr.org/.
[7] Terrence Parr, "The Definitive ANTLR 4 Reference", The Pragmatic Programmers, LLC., 2012.
[8] GitHub. (May 28). "antlr/grammars-v4" [Online]. Available: https://github.com/antlr/grammars-v4/blob/master/java/Java.g4.