ISBN: 978-981-18-5852-9
WCSE 2022 Spring Event: 2022 9th International Conference on Industrial Engineering and Applications
doi: 10.18178/wcse.2022.04.176

Model Checking Java Programs with JPS

Xinfeng Shu[1], Gege Quan[1]

[1] School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an 710121, China

shuxf@xupt.edu.cn, quan15735659315@stu.xupt.edu.cn

Abstract. In order to solve the verification problem of Java programs, a novel model checking approach with JPSL (Java Property Specification Language) is advocated. To this end, the JPSL is defined for describing the desired properties of Java programs with a specific format of annotation mixed into the source code, and the technique for automatically converting the JPSL properties into their equivalent PPTL (Propositional Projection Temporal Logic) formulas is formalize, which in turn can be directly used to model checking Java programs with MSV tool. In addition, an example is given to illustrate how the approach works. The approach provides a convenient and powerful way for engineers to verify Java programs, and helps to improve the quality of the software system.

Keywords: JPSL, Java, program verification, model checking

1. Introduction

Java [1], a famous object-oriented programming language, has been widely used in various areas of software development. Facing the generous software written in Java, how to ensure their correctness and reliability is of grand challenge to computer scientists as well as software engineers. To solve the problem, software testing has been developed for many years and a variety of tools has been developed to verify software systems with success. However, the method has its innate limitation, i.e., it can only prove the presence of errors but never their absence. In contrast, formal verification, which is based on the mathematical theories, can prove the correctness of the software and become an important means to verify software systems.

Model checking [2] is an important formal verification technique, which an exhaustively search each execution path of the system model to be verified, and check whether the desired property holds. In the early days, the research on model checking mainly focuses on verifying the analysis and design models of hardware and software systems with the classic tools such as SPIN [3] or NuSMV [4]. The kernel process of the verification is to model the system with a specific modeling language (e.g., Promela [5] or SMV [6]) and specify the properties of the system with a temporal logic, e.g., Linear Temporal Logic (LTL) [7] or Computation Tree Logic (CTL) [8]. The works usually need to be finished by verifiers manually, for complex system, it is very difficult to guarantee their correctness.

In recent years, some methods for model checking C, C++ and Java programs have been advocated, and a number of model checking tools have been developed (e.g., SLAM [9], BLAST [10], MAGIC [11], ESBMC [12] and JavaPathFinder [13]) to verify device drivers, communication protocols, real-time operating system kernel with success. Besides, the available tools can only check the safety property and dead lock of the system, but cannot verify the liveness property.

In addition to the above methods, model checking C and Java programs [14,15] with MSVL (Modeling, Simulation and Verification Language) are also important verification approaches. MSVL [16,17], an execution subset of Projection Temporal Logic (PTL) [18] and process-oriented logic programming language, is a useful formalism for specification and verification of concurrent and distributed systems. It provides a rich set of data types, data structures as well as powerful statements [19,20]. Besides, MSVL supports the function mechanism to model the complex system [21]. Besides, PPTL [22,23], the propositional subset of PTL, has the expressiveness power of the full regular expressions [24], and can describe more complex system properties than LTL, such as the closure property of repeated execution.

While model checking, specifying the system properties is one of the two kernel works. The available methods directly use temporal logic (e.g., LTL, CTL or PPTL) formulas to describe the properties. For software engineers, they have good understanding of the program designment, but, they usually have no enough temporal logic knowledge to specify the properties correctly. The problem greatly affects the promotion and application of model checking Java programs in industry.

To solve the problem above, in this paper, we extend the MSVL based model checking approach for Java programs [15] by introduces a specific language, named Java Property Specific Language (JPSL), to describe the properties of the program. JPSL is a specific format of program annotation, which takes the Pre-Condition, Post-Condition like familiar style of software engineers to describe properties of Java programs instead of complex logic symbol. Moreover, the desired properties of Java classes, functions, code fragments, and program statements are specified as the JPSL statements labeled on the corresponding sections of source code. Furthermore, the conversion method and related techniques from JPSL statements to PPTL formulas are presented, and the result obtained can be used as the properties' specification to model checking Java program directly.

The rest of this paper is organized as follows. In the next section, PPTL and Java language are briefly introduced respectively. In Section 3, JPSL and its specific labeling position are defined. In Section 4, the rules for converting JPSL to PPTL are defined and the related techniques are introduced. In Section 5, an example is given to illustrate how the method works. Finally, the conclusion is given in Section 6.

2. Preliminaries

2.1. Propositional Projection Temporal Logic

Propositional Projection Temporal Logic (PPTL) is the propositional subset of PTL which supports both finite and infinite models. In this section, the syntax and semantics of PPTL are briefly introduced. More details can be found in literature [21].

2.1.1. Syntax

Let ϕ be a finite set of atomic propositions, and *B={true, false}* the boolean domain. The formula P of PPTL are inductively defined as follows:

$P ::= p \mid \neg P \mid P_1 \wedge P_2 \mid \bigcirc P \mid P^+ \mid (P_1,...,P_m)\, prj P$

where $p \in \phi$ is an atomic proposition; $\bigcirc$ (next), + (chop-plus) and *prj* (projection) are temporal operators, and ¬ and $\wedge$ are identical to those in the classical propositional logic. A formula is called a state formula if it contains no temporal operators. The conventional constructs *true*, *false*, $\wedge$, $\rightarrow$ as well as $\leftrightarrow$ are defined as usual.

2.1.2. semantics

A state *s* over ϕ is a mapping from ϕ to *B*, i.e., *s*: $\phi \rightarrow B$ .We use notation *s*[*p*] to denote the valuation of *p* at state *s*. An interval (i.e., model) σ is a non-empty sequence of states $\sigma = < s_0,...,s_{|\sigma|} >$, which $|\sigma|$ denotes the length of σ and is ω if σ is infinite, or the number of states minus one if σ is finite. Let N_0 be the set of non-negative integers and $N_\omega = N_0 \cup \{\omega\}$, we extend the comparison operators, =, <, ≤, to N_w by considering $\omega = \omega$, and for all $i \in N_0$, $i < \omega$. Moreover we define $\preceq$ as $\leq - \{(w, w)\}$ .We use notation $\sigma_{(i,...,j)}$ to mean that a subinterval $< s_i,...,s_j >$ of σ with $0 \leq i \preceq j \leq |\sigma|$. The concatenation of a finite interval $\sigma = < s_0,...,s_{|\sigma|} >$ with another interval $\sigma' = < s'_0,...,s'_{|\sigma'|} >$ (may be infinite) is denoted by $\sigma \bullet \sigma'$ and $\sigma \bullet \sigma' = < s_0,...,s_{|\sigma|}, s'_0, \ldots, s'_{|\sigma'|} >$.

2.2. Java Programming Language

Java [1] is a popular object-oriented programming language with the feature "write once, run anywhere", and hence has been widely used in developing web and mobile application, big dada processing, etc. It not only supports the object-oriented mechanism, but also provides multi-thread, socket and interface programming. In this paper, we only focus on the object-oriented part of java. In the following, we briefly introduce the grammar of the subset of Java language to be verified.

2.2.1. Data type: The data types of Java programming language are divided into two categories, i.e., basic data types and reference data types. Basic data types include character (char), integer (byte, short, int, long) and floating point (float, double), boolean (boolean). Reference data types include class, interface(interface), array and so on.

2.2.2. Expression: Let d be a constant and x be a variable respectively. The arithmetic expressions e and boolean expressions b of Java are inductively defined as follows:

$$e ::= d \mid x \mid e_1\ op_1\ e_2 (op_1 ::= + \mid - \mid * \mid / \mid \% \mid ++ \mid --)$$

$$b ::= true \mid false \mid !b \mid e_1\ op_2\ e_2 (op_2 ::= > \mid < \mid == \mid >= \mid <= \mid !=) \mid$$

$$b_1\ op_3\ b_2 (op_3 ::= \&\&, \mid\mid)$$

where op_1 denotes the traditional arithmetic operators, op_2 are the relational operators and op_3 the logical operators.

2.2.3. Elementary statement: Let type be a data type, x be a variable, d, d_1, ..., d_n be constants, and obj be an object. The elementary statements of Java are inductively defined as follows:

a. Declaration statement type x | type x=e | $dcls_1$, $dcls_2$

b. Assignment statement x=e / obj.attr=e |

c. Function call statement obj.fun*(*e_1,...e_n*)*

d. Compound statement *{*s*}*

e. Sequential statement s_1*;*s

f. If statement if*(*b*){*s*}* | if*(*b*){*s_1*}*else*{*s_2*}*

g. For statement for*(*dcls*;* b*;* e*){*s*}*

h. While statement while*(*b*){*s*}*

i. Do-While statement do*{*s*}*while*(*b*)*

j. Switch statement switch*(*x*){*case d_1 *:* s_1*;*

*[*break*]; ... ; [*default *:* s*]}*

where *dcls*, $dcls_1$ and $dcls_2$ are any declaration statements; *fun* is a member function of *obj* with $n(n \geq 0)$ parameters, and *attr* is an attribute of *obj*; e, e_1, ... , e_n are expressions; s, s_1, ... , s_n can be any statements.

2.2.4. Class definition: Java is an object-oriented programming language supporting only single inherence, i.e., each class has at most one super class.

3. Java Property Specification Language

In order to reduce the difficulty of describing the desired properties of the Java programs, in this paper, we define a specific language named Java Property Specific Language (JPSL) for software engineers. JPSL employs the assertions, pre/post conditions like familiar way of engineers to specify the Java program properties as JPSL statements associated with the program unit mixed in source code. Moreover, JPSL uses specific keywords instead of complex temporal logic operators to easily describe the temporal properties of Java programs such as liveness, security, and fairness.

3.1. Syntax and Semantics

The grammar definition of JPSL is shown in Table II, among which the JPSL statement *JPSL_Stmt* is in the form of Java program annotations and identified by the keyword @JPSL; keywords SEC_BEGIN and SEC_END are used to mark the beginning and ending of a code fragment respectively that must appear in pairs. The meanings of the JPSL keywords and operators are shown in Table III and Table IV respectively.

TABLE I. JPSL DEFINITION

Category	***Definition***		
JPSL Statement	*JPSL_Stmt*	::=	/* @JPSL [ADD | REP] *Prop* */ | /* @JPSL SEC_BEGIN *Prop* */ | /* @JPSL SEC_END */
Property Statement	*Prop*	::=	*Pred* | SEQ($Prop_1$; …; $Prop_n$) | REPEAT($Prop_1$) | SOMETIMES ($Prop_1$) | ALWAYS ($Prop_1$) | PRE ($Prop_1$) | POST ($Prop_1$) |!$Prop_1$ | $Prop_1$&&$Prop_2$ | $Prop_1$ || $Prop_2$
Predicate	*Pred*	::=	Exp_1(< | <= | == | > | >=)Exp_2
Expression	*Exp*	::=	*const* | *v* | *obj.attr* | *this.attr* | *class.attr* | Exp_1 (+ | - | * | / | %) Exp_2

TABLE II. JPSL KEYWORD

Keyword	***Implication***	***Keyword***	***Implication***
@JPSL	JPSL Statement identification	ADD	Additional property
REP	Substitution property	SEC_BEGIN	Beginning tag of a code fragment
SEC_END	Ending tag of a code fragment	SEQ	Properties hold in turn
REPEAT	Property repeatedly holds	SOMETIMES	Property holds at some time
ALWAYS	Property always holds	PRE	Property holds at the beginning
POST	Property finally holds		

TABLE III. JPSL OPERATORS

Priority	***Operator***	***Meaning***	***Associativity***
1	/* */	annotation operator	from left to right
2	( ), ;	parentheses, semicolons	from left to right
3	!	logical negation	from right to left
4	*, /, %	multiply, divide and take the remainder	from left to right
5	+, -	add, subtract	from left to right
6	<, <=, ==, >, >=	less than, less than or equal to, greater than, greater than or equal to	from left to right
7	&&	logical and	from left to right
8	||	logical or	from left to right

3.2. Specifying Properties of Java Programs with JPSL

JPSL specifies the expected properties of Java programs by JPSL statements attached to classes, functions, code fragments and program statements of Java source code with the following rules.

Rule 1: For the property that all the objects of a class must follow during their lifetime, we describe the property with a JPSL statement and insert it before the definition of the class. This kind of JPSL statement is also referred to as “class property statement”;

Rule 2: For the property that a function of a class must enjoy while execution, we describe the property with a JPSL statement and insert it before the definition of the function. This kind of JPSL statement is also referred to as "function property statement";

Rule 3: For the property that a code segment of a function must keep while execution, we specify the property with a JPSL statement and insert the correspond "SEC_BEGIN" and "SEC_END" before and after the code segment respectively. This kind of JPSL statement is also referred to as "code segment property statement";

Rule 4: For the property that a statement of a function must meet before execution, we specify the property with a JPSL statement and insert it before the statement. This kind of JPSL statement is also referred to as "assertion statement".

4. Conversion of JPSL to PPTL

The properties described in JPSL cannot be used to model checking Java programs directly, therefore, we need to convert the JPSL statements embedded in Java programs into properties specified in PPTL formulas. This section first presents the specific conversion process from JPSL to PPTL, and then gives the calculation rules for constraint positions and the conversion rules from JPSL statements to PPTL formulas respectively.

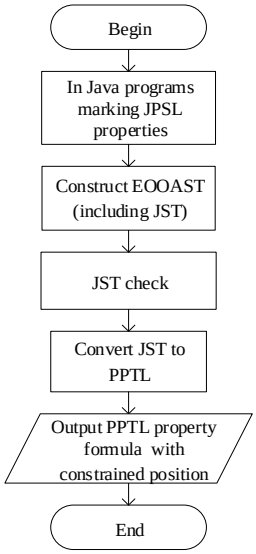

Fig. 1. Flow chart of converting JPSL into PPTL property specification

The conversion process from JPSL to PPTL is depicted in Figure 1. While verifying a Java program, we need first specify the desired properties of the programs with use JPSL, and then perform lexical and grammatical analysis on the program with JPSL statements, and construct an extended object-oriented abstract syntax tree (EOOAST) to describe the Java program itself as well as the syntax tree (JST) of JPSL statements. Subsequently, check the integrity and consistency of the JST in EOOAST, and convert each JST in EOOAST into a PPTL formula with constraint position.

4.1. Constructing EOOAST

The EOOAST for describing the syntax of the Java program itself and the JPSL statements is depicted in Figure 2, and the extended hierarchy syntax diagram (EHSD) for describing the syntax of a function is shown in Figure 3. In EOOAST, each class node is composed of an attribute node set *AttrSe*t, a JPSL statement syntax tree JST, and a function node set *FuncSet*, among which *AttrSet* contains n ($n \geq 1, n \in N^+$) attribute nodes to keep all the attributes of the class, JST saves the syntax tree of the corresponding JPSL statement, and *FuncSet* consists of n extended hierarchy syntax diagrams of the member functions with JSTs attached to the corresponding function node or statement node.

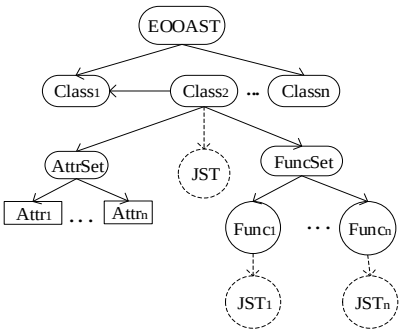

Fig. 2. Extended object-oriented abstract syntax tree

4.2. Checking the Integrity and Consistency of JST

After the EOOAST of the Java program has been created, we need further check the integrity and consistency of each JST in EOOAST. The basic strategy is to identify whether JST complies with JPSL grammar and whether the reference variables is correct, which the class property statement can only refer to the direct attributes or inherited attributes of the class, whereas the other three kind of JPSL statements can refer to the class attributes, the formal parameters and the local variables of the functions.

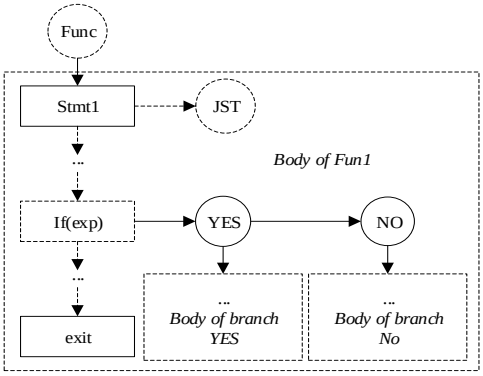

Fig. 3. Extended hierarchy syntax diagram of a member function

4.3. Converting JST into PPTL Formula

For converting a JST of JPSL statement in the EOOAST into PPTL formula, the result is a triple (*scope*, *pptl*, *map*), which *scope* represents the constraint range of JPSL statement; *pptl* represents the PPTL formula corresponding to the JPSL property; *map* is the set of mapping from atomic propositions in *pptl* and the predicates which stand for. The calculation rules for *scope*, *pptl* and *map* are as follows:

4.3.1. Calculation rules for constraint range

Rule 1: If JST is a class property statement marked on the class node *cls*, the constraint range *scope* is *cls*;

Rule 2: If JST is a function property statement marked on the member function node *fun* of class *cls*, the constraint range *scope* is *cls*:*fun*;

Rule 3: If JST is a code segment property statement marked from line num_0 to line num_1 in the function *fun* of class *cls*, the constraint range *scope* is *cls*:*fun*:num_0:num_1;

Rule 4: If JST is an assertion statement marked on line *num* in the function *fun* of class *cls*, the constraint range *scope* is *cls*:*fun*:*num*.

4.3.2. Calculation rules for PPTL formula and mapping set

Rule 1: If JST is a predicate e_1 [<|<=|==|>|>=] e_2, the result PPTL formula *pptl* is a fresh atomic proposition *p*, and the mapping set *map* is {*p*: e_1 [<|<=|==|>|>=] e_2};

Rule 2: If JST is a property statement SEQ($Prop_1$; ... ; $Prop_n$), first convert each $Prop_i$ ($1 \le i \le n$) respectively, let the result be $pptl_i$ and map_i, the final result PPTL formula *pptl* is $pptl_1$;...;$pptl_n$, and the mapping set *map* is $map_1 \cup ... \cup map_n$;

Rule 3: If JST is a property statement REPEAT ($Prop_1$), first convert $Prop_1$, let the result be $pptl_1$ and map_1, the

final result PPTL formula *pptl* is $(pptl_1)$+, and the mapping set *map* is map_1;

Rule 4: If JST is a statement SOMETIMES($Prop_1$), first convert $Prop_1$, let the result be $pptl_1$ and map_1, the final result PPTL formula *pptl* is $\Diamond pptl_1$, and the mapping set *map* is map_1;

Rule 5: If JST is a statement ALWAYS ($Prop_1$), first convert $Prop_1$, let the result be $pptl_1$ and map_1, the final result PPTL formula *pptl* is $\Box pptl_1$, and the mapping set *map* is map_1;

Rule 6: If JST is a statement PRE($Prop_1$), first convert $Prop_1$, let the result be $pptl_1$ and map_1, the final result PPTL formula *pptl* is $pptl_1$, and the mapping set is map_1;

Rule 7: If JST is a statement POST($Prop_1$), first convert $Prop_1$, let the result be $pptl_1$ and map_1, the final result PPTL formula *pptl* is $\Box(\varepsilon \rightarrow pptl_1)$, and the mapping set is map_1;

Rule 8: If JST is a statement $!Prop_1$, first convert $Prop_1$, let the result be $pptl_1$ and map_1, the final result PPTL formula *pptl* is $\neg pptl_1$, and the mapping set *map* is map_1;

Rule 9: If JST is a statement $Prop_1 \&\& Prop_2$, first convert $Prop_1$ and $Prop_2$, let the results be $pptl_1$, map_1, $pptl_2$, and map_2 respectively. The final PPTL formula *pptl* is $pptl_1 \wedge pptl_2$, and the mapping set *map* is $map_1 \cup map_2$;

Rule 10: If JST is a statement $Prop_1 \| Prop_2$, first convert $Prop_1$ and $Prop_2$ respectively, let the results be $pptl_1$, map_1, $pptl_2$ and map_2 respectively. The final PPTL formula *pptl* is $pptl_1 \vee pptl_2$, and the mapping set *map* is $map_1 \cup map_2$.

5. Verification Case

In following, we give an example to illustrate how our method works in verifying a Java program. The 3*x*+1 conjecture is a well-known but unsolved problem in number theory, which asserts that for any given positive integer *x*, if *x* is an even number, let *x*=*x*/2, otherwise let *x*=*x**3+1. If we repeatedly applying the calculation rule to *x*, the value of *x* must eventually be 1. The 3*x*+1 conjecture can be described as the Java program in Figure 4.

Firstly, we employ JPSL to label the desired properties of the Java program in source code:

1) The attribute *value* of class Quess3X1 is used to store the value of the positive integer *x* in the "3*x*+1 problem", which must always be greater than 0 during the computation. The property described in JPSL is the class statement "/*@JPSL ALWAYS (*value*>0) */" labeled as the annotation in front of the Quess3X1 class (see line 1).

2) Before calling the method *run* to calculate according to the rule of "3x+1", the value of *value* must be greater than 1, which described in JPSL is a function statement "/*@JPSL PRE (*value* >1) */". The property is labeled as the annotation in front of the method *run* (see line 16).

```
1   /*@JPSL ALWAYS (value >0) */
2   public class Quess3X1 {
3     private  int  value=1;
4     public  void  setValue ( int  number ) {
5       if (number < 1)
6         value = 1;
7       else
8         value = number;
9     }
10    public  int  getValue( ) {
11      return value;
12    }
13    public  boolean  isEven( ) {
14      return value % 2==0;
15    }
16    /*@JPSL PRE(value>1) */
17    public  void  run( ) {
18     /*@JPSL SEC_BEGIN  POST(value =1)*/
19     while(1 < value) {
20       if( isEven( ) )
21         value = value / 2;
22       else
23         value = value * 3 + 1;
24     }
25     /*@JPSL SEC_END*/
26    }
27    public static void main(String[]  args) {
28      Quess3X1 demo = new Quess3X1( );
29      System.out.println("Input a positive number:");
30      Scanner in = new Scanner(System.in);
31      int x = in.nextInt();
```

```
32              demo.setValue(x);
33              demo.run();
34              x = demo.getValue();
35              System.out.println(x);
36            }
37        }
```

Fig. 4. Java program with JPSL properties for 3x+1 conjecture

3) After execution of the code segment *while*(1<*value*) {…} in the function *run*, the attribute *value* must equal 1, which is described as a code segment property statement "/* @JPSL SEC_BEGIN POST (*value*=1)*/" and /*@JPSL SEC_END*/" labeled as the annotations before (lines 18) and after (line 25) the code segment respectively.

Secondly, perform the lexical and syntax analysis on the program, and construct the EOOST. For simplicity, we only give the extended hierarchical syntax diagram of function *run* .The JST of each JPSL is similar to the syntax tree of an arithmetic express and hence is omitted here.

Thirdly, check the integrity and consistency of the JST in the EOOAST. Obviously, the three JPSL statements in Figure 4 conform to the grammar of JPSL, and they only refer to the attribute *value* of class Quess3X1, so the check is successfully passed.

Subsequently, convert each JST into the desired property expressed in PPTL according to the calculation rules as follows:

1) For the JST in line 1, according to Rule 1 of constraint range calculation and Rule 1 of PPTL formula and mapping set calculation, the final result is (Quess3X1, □p1, {p1: value>0});

2) For the JST in line 16, according to Rule 2 of constraint range calculation and Rule 6, Rule 1 of PPTL formula and mapping set calculation, the final result is (Quess3X1:run, p2, {p2: value>1});

3) For the JST in line 18 and line 25, according to Rule 3 of constraint range calculation and Rule 7, Rule 1 of PPTL formula and mapping set calculation, the final result is (Quess3X1:run:18:25, □(ε →p3), {p3: *value*=1}).

Finally, transform the Java program for "3*x*+1" conjecture into its equivalent MSVL program model with the technique in literature [15], and the result is shown in Figure 5. We now can verify the Java program by indirectly model checking the corresponding MSVL program with the PPTL properties obtained above using the MSV tool. Here we only give the example of model checking the property (Quess3X1, p1, {p1: value>0}), the other properties can be verified in a similar way.

Corresponding to the property (Quess3X1, p1, {p1: value>0}), the original JPSL statement is constraint on class Quess3X1, by the semantics of class property, all the objects of the class must abide by the property in their lifetime. According to the transforming rules from Java program into MSVL program, the constraint is passed to the MSVL *struct Ques3X1*, and all the variables of struct Ques3X1 must conform to the PPTL property in their application scopes. Moreover, the only variable of struct Ques3X1 is *demo* in function *Ques3X1_main*, and the attribute value is embedded as the member *value* of *demo*. Thus, the PPTL formula for model checking with MSV tool is definition as

```
</
    define p1: demo.value = 1 ;
     alw (p1)
/>
```

We model check the MSVL program on the MSV tool with the input integer 111, an empty LNFG with no edge is produced as shown in Figure.6. Thus, the property holds.

```
struct Ques3X1{
        int value};
function Ques3X1_getValue (struct Ques3X1 * this, int * Ret) {
        *Ret<==this ->value and skip};
function Ques3X1_setValue (struct Ques3X1 * this, int num) {
      if (num<1) then
            this ->value :=1 and skip
      else
           this->value := num and skip};
function Ques3X1_isEven (struct Ques3X1 * this, boolean * Ret) {
          *Ret := this->value%2==0 and skip};
function Ques3X1_run (struct Ques3X1 * this) {
      frame (Ret) and (
            while (this -> value >1) {
                  boolean Ret and Ret:=Ques3X1_isEven(this) and skip ;
                  if (Ret) then
                        this ->value := this -> value/2 and skip
                  else
                        this ->value := this ->value*3+1 and skip}) };
};
function Ques3X1_main () {
      frame (demo, x) and (
            struct Ques3X1 demo and skip;
            int x and skip;
            Output ("Input a positive number:" ) and skip;
            input(x) and skip;
            Ques3X1_setValue (&demo, x) and skip;
            Ques3X1_run(&demo) and skip;
            x :=Ques3X1_getValue(&demo) and skip;
            output (x) and skip)};
Ques3X1_main ()
```

Fig. 5. MSVL program obtained for the Java program of 3x+1 conjecture

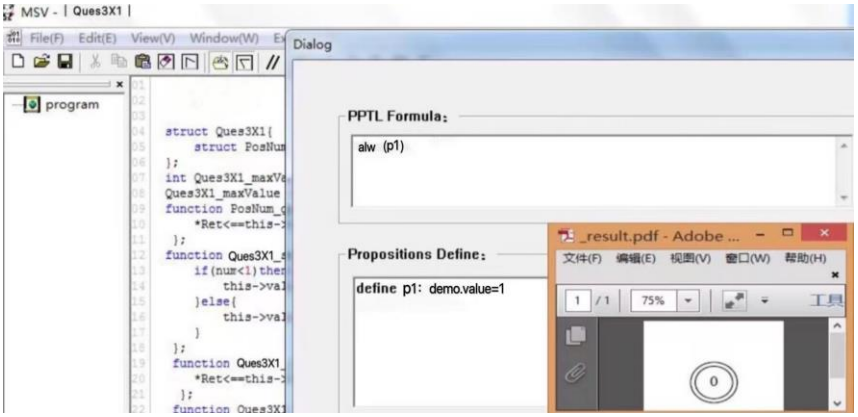

Fig. 6. Verification result of the 3x+1 program

6. Conclusion

In this paper, we present a novel model checking approach for Java programs by specifying the desired properties of the system with JPSL statements labeled on the elements of Java source code. Compared to the existing model checking methods of Java programs, JPSL can easily be mastered by software engineers to specify the system properties while programming. Moreover, the method proposed fully utilize the expressiveness power of PPTL to model check more properties such as safety and liveness, etc.

7. References

[1] Arnold K, Gosling J, Holmes D. Java Programming Language (4th Edition). Addison-Wesley Professional, 2005.

[2] Baier C, Katoen J P. Principles of Model Checking. The MIT Press, 2008.

[3] Holzmann G J. Software model checking with SPIN. Advances in Computers, 2005,65:77-108.

[4] Cimatti A, Clarke E, Giunchiglia F, et al. NUSMV: a new symbolic model checker.International Journal on Software Tools for Technology Transfer, 2000,2(4):410-425.

[5] Staroletov S M. A Formal Model of a Partitioned Real-Time Operating System in Promela. Proceedings of the Institute for System Programming of RAS, 2021,32(6):49-66.

[6] Feng Liu, Zhoujun Li, Mengjun Li, et al. Security protocol model checking based on SMV. Computer Engineering and Science,2004,26(002):28-31,62.

[7] Babenyshev S, Rybakov V. Unification in linear temporal logic LTL. Annals of Pure and Applied Logic, 2011,162(12):991-1000.

[8] R.Cavada, A.Cimatti,C.A.Jochim. NuSMV 2.5 User Manual. http://nusmv.fbk.eu/ NuSMV/userman/v25/nusmv.pdf

[9] Thomas Ball and Sriram K.Rajarnani. The SLAM Project: Debugging System Software via Static Analysis. POPL 2002:1-3.

[10] Homas A. H., Ranjit J., Rupak M. and Gregoire S. Software Verification with BLAST. SPIN 2003, LNCS 2648: 235-239.

[11] Chaki S, Clarke E, Groce A, et al. Modular verification of software components in C. International Conference on Software Engineering, 2003:385-395.

[12] Cordeiro L, Fischer B. Context-Bounded model checking with ESBMC 1.17. Lecture Notes in Computer Science 7214:534-537(2012).

[13] Brat G, Havelund K, Visser W. Java PathFinder-Second Generation of a Java Model Checker, 2000

[14] Meng Wang, Cong Tian, Nan Zhang, Zhenhua Duan, Chenguang Yao. Translating Xd-C programs to MSVL programs. Theoretical Computer Science, 2020, vol.809: 430-465.

[15] Xinfeng Shu, Na Luo, Bo Wang, Xiaobing Wang, Liang Zhao. Model Checking Java Programs with MSVL. SOFL+MSVL 2018: 89-107.

[16] Duan, Z., Yang, X., Koutny, M.. Framed temporal logic programming. Sci. Comput. Program. 70(1):31-61 (2008).

[17] Zhenhua Duan. Temporal logic and temporal logic programming. Science Press, 2005.

[18] Xinfeng Shu, Zhenhua Duan, Hongwei Du. A decision procedure and complete axiomatization for projection temporal logic. Theor. Comput. Sci. 819: 50-84 (2020).

[19] Xiaobing Wang, Cong Tian, Zhenhua Duan, Liang Zhao. MSVL: A Typed Language for Temporal Logic Programming. Frontiers of Computer Science. DOI:10.1007/s11704-016- 6059-4.

[20] Xinfeng Shu, Zhenhua Duan. Extending MSVL with Semaphore. COCOON 2016: 599-610

[21] Nan Zhang, Zhenhua Duan, Cong Tian. A mechanism of function calls in MSVL. Theoretical Computer Science, 2016, Vol.654:11-25.

[22] Xinfeng Shu, Nan Zhang, Xiaobing Wang, Liang Zhao. Efficient decision procedure for propositional projection temporal logic. Theor. Comput. Sci. 838: 1-16 (2020).

[23] Nan Zhang, Zhenhua Duan, Cong Tian. A complete axiom system for propositional projection temporal logic with cylinder computation model. Theor. Comput. Sci. 609: 639-657 (2016)

[24] Cong Tian, Zhenhua Duan. Expressiveness of propositional projection temporal logic with star. Theor. Comput. Sci. 412(18): 1729-1744 (2011)