Predicate Interpretation Analysis Based on Soot

Chunrong Fang, Qingkai Shi, Yang Feng, Zicong Liu, Xiaofang Zhang, Baowen Xu*
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing 210093, China
*Email: bwxu@nju.edu.cn

***Abstract*—Symbolic execution maintains a path condition pc for every possible path of a program. It is challenging to construct a pc if some complex issues are involved in the path. A predicate interpretation *pi* is a subexpression of a pc and a pc of a path is a conjunction of all *pi*s in the path. Predicate interpretation has been widely used in theoretical analysis on domain testing and related fields. It recently emerges new impact on software testing by using partial path constraints to generate test data. In this paper, we propose an approach to produce *pi*s in a program. A tool for predicate interpretation analysis for Java programs is implemented based on the data-flow framework of Soot. Most of Java features can be handled in our tool. Moreover, a formal rule of predicate interpretation analysis is presented for more applications in the future. The experimental results show that our tool can produce *pi*s of a program effectively and efficiently.**

I. Introduction

Predicates play a key role in programs to fulfill functionalities with different execution paths. Predicate analysis on paths have shown increasing significance in software testing and debugging. Symbolic execution is a commonly used technique to construct path conditions (pc) as constraints, which hold the execution of a path in a program [7]. However, there are some challenges to make it practical in industry. One of the challenges is to effectively construct a pc of a path involving some complex issues in a program [2].

Predicate interpretation (pi) [10] is a subexpression of a pc. That is, a pi is a partial path constraint and a pc is a conjunction of all pis in the path. A pi is a predicate with only input variables. pis on a branch may be different because it may appear on different execution paths. pi has been widely used in theoretical analysis on domain testing and related fields. Recently, pi emerges new impact on software testing by using partial path constraints to generate test data, as pis are more practical than a pc involving some complex issues. This inspires us to build an effective tool for predicate interpretation analysis on modern programming languages, such as Java.

Data-flow analysis is an important method of program static analysis [6], which provides the theoretical basis of predicate interpretation analysis. Soot [8], a well-known framework for analyzing or optimizing Java byte-code, based on which many analysis tools have been implemented[1]. A key feature of the Soot framework is its excellent support for implementing intra-procedural data-flow analysis. In addition, Soot includes call-graph information and pointer analysis framework, which help deal with some complex issues in our approach.

[1]http://www.sable.mcgill.ca/soot/

In this paper, we propose an approach of predicate interpretation analysis based on data-flow framework of Soot. Our approach collects appropriate predicates respectively and replaces variables according to assignment statements along with the corresponding data flows. A novel feature of our approach is using loop induction to reduce the search space and improve the speed of analysis. A formal rule of predicate interpretation produced by the tool is presented for further uses. Additionally, a preliminary experiment was conducted. The experimental results show that our tool can produce pis of programs effectively and efficiently.

II. Approach

A. Data-flow Framework

In this section, we adopt a unified data-flow analysis framework based on lattice [6] to support PI analysis. The framework (D, V, $\wedge$, F) [1] is described as follows. (1) A direction of the data flow D, which is either FORWARDS or BACKWARDS. (2)A semi-lattice, including a domain of values V and a merge operator $\wedge$. A semi-lattice has a top element, denoted $\top$, such that for all $x \in V$, $\top \wedge x = x$. (3) A family F of transfer functions from V to V. A function in F, such as f_s, is a transfer function of the statement s.

We denote the data-flow values before and after each statement s by IN[s] and OUT[s], respectively. Given a data-flow graph and a backward data-flow problem, the unified algorithm [1] is shown in Figure 1.

```
initialize a value set: $v$;
IN [EXIT] = $v$;
for (each statement $s$ other than EXIT) IN[$s$] = $\top$;
while (changes to any IN occur)
        for (each statement $s$ other than EXIT) {
              OUT[$s$] = $\wedge_{m \text{ a successor of } s}$ IN[$m$] ;
              IN[s] = $f_s$(OUT[$s$]);
        }
```

Fig. 1. Algorithm of backward data-flow

Data-flow values of every point in the data-flow graph are initialized firstly. An iteration from the exit to the entry of the graph is followed by the initialization. Each time we pass by a statement s. The transfer function of the statement will be used. If two data-flow values meet at a point, they will be merged in accordance with the merge operation.

DOI reference number: 10.18293/SEKE2017-052

B. Predicate Interpretation

A predicate interpretation (pi) is a predicate that only contains parameters of a program [10]. If a predicate can be represented by $expression \Theta 0$, where Θ is a relational operator and $expression$ is a common arithmetic expression, the corresponding pi is $expression$ only with parameters $\Theta 0$.

PI analysis uses BACKWARDS data-flow technique. The value domain V is the set of predicates, and the merge operation $\wedge$ is union $\cup$. Some key points of data-flow framework for PI analysis are summarized in Table 1. It is a monotonic and distributive [6] data-flow framework which implies the solution of the problem is the maximum fixed point solution to the data-flow equations. We can apply these framework elements in Table 1 to the algorithm in Figure 1 to obtain the result of PI analysis.

TABLE I
PREDICATE INTERPRETATION ANALYSIS

Elements	**PI Analysis**
Domain	Set of predicates
Direction	Backwards
Transfer Function	(1)(2)(3)
Boundary	IN[EXIT]=Ø
Merge ($\wedge$)	$\cup$
Data-flow Equations	IN[s]= f(OUT[s]), OUT[s]= $\wedge_{p,succ(s)}$IN[p]
Initialize	IN[s]=Ø, for each s.

Transfer functions in Table 1 are important for PI analysis. We explain the transfer functions in detail as follows.

(1) If a statement s is a branch statement, the transfer function of s will be

$$f_s(x) = x \cup gen_s \tag{1}$$

where x is the pi set before the statement and gen_s is a set of predicates in the statement s.

(2) If a statement s is an assignment statement of variable v. Suppose the statement is $v = g(v, y_0, y_1, y_2, \cdots)$, where g is a function about variables v and $y_i, i = 0, 1, 2, \cdots$. The transfer function is

$$\forall pi(v) \in x,\ f_s(pi(v)) = pi(g(v, y_0, y_1, y_2, \cdots)) \tag{2}$$

which means variable v in every predicate in x, the pi set before statement s, will be substituted with $g(v, y_0, y_1, y_2, \cdots)$. In the equation, $pi(v)$ means a predicate containing the variable v.

(3) For others, transfer function is an identity function I about the domain value x, such that

$$I(x) = x \tag{3}$$

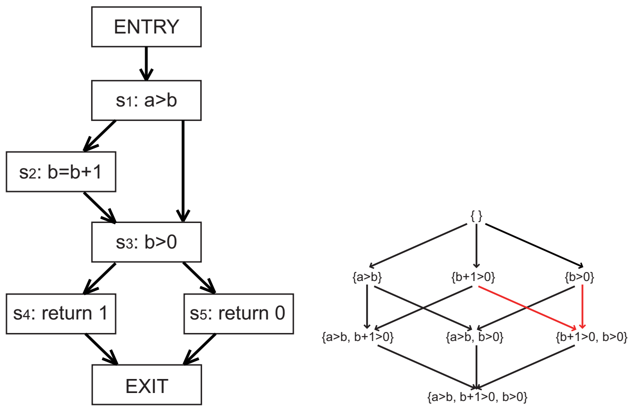
C. An Example

We introduce a simple example to explain our approach in detail. Figure 2(a) shows a control-flow graph.

If we define a partial order $\leq$ of a semi-lattice, such that

$$\forall x, y \in V,\ x \leq y\ iff.\ x \wedge y$$

the partial order $\leq$ of PI analysis semi-lattice will be $\supseteq$ because the merge operation is $\cup$. Following by the definition, we can draw the domain V as a lattice diagram in Figure 2(b).



(a) A control-flow graph (b) Lattice diagram

Fig. 2. An Example

The edges are all downward. From the diagram, we can get the result of merge operation easily.

Before the statement s_3: $b > 0$, OUT[s_5]={} due to the initial condition and using the identity transfer function. After s_3, IN[s_3]=f_{s_3}(OUT[s_3]) $= \{\} \cup \{b > 0\} = \{b > 0\}$. It uses the transfer function (1), and OUT[s_2]=OUT[s_1]=IN[s_3]=$\{b > 0\}$.

When analysis passes assignment statement s_2, using transfer function (2), we replace variable b with $b + 1$, such that IN[s_2] = $\{b + 1 > 0\}$. IN[s_2] and IN[s_3] will meet at the exit of statement s_1, where we use the merge operation according to the red edges from Figure 2(b), such that, OUT[s_1]=IN[s_2] $\cup$ IN[s_3]= $\{b + 1 > 0, b > 0\}$.

Finally, we can obtain the resulting set of pis, which equals to IN[ENTRY]=$\{a > b, b + 1 > 0, b > 0\}$.

III. IMPLEMENTATION

The framework of our tool for PI analysis based on Soot is illustrated in Figure 3. It is made up of four main parts:(1) **Jimple Parser** transforms Java byte code to Jimple code, which is provided by Soot. (2) **Transformer** transforms the input Jimple code, with our two new transformers: `MethodsInliner` for inlining some user-defined methods. `StaticFieldInitializer` for initialize some static fields. (3) **Data-flow Analysis** is built on the data-flow framework of Soot. (4) **Filter** is used to filter some redundant expressions.

A. Transformer

Transformers accept Jimple codes of a program body as input, and then transform them by two new body transformers. The first one is `MethodsInliner`, which is used to inline methods. In default case, our tool will inline all user-defined methods and Java library methods will remain. The second one is `StaticFieldsInitializer`, which is used to initialize static fields of a class.

In most cases, `MethodsInliner` firstly inlines all user-defined methods, and obtain some classes whose static fields have been used in the program body.

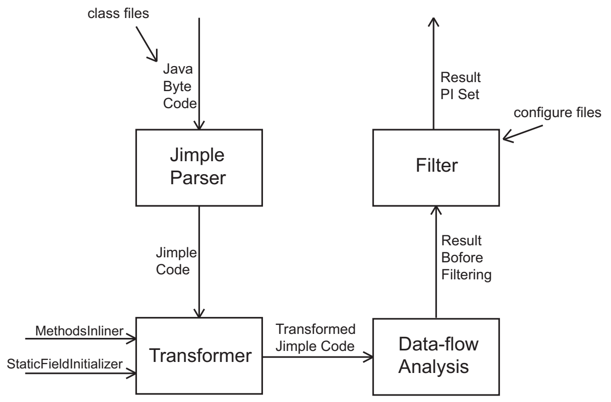
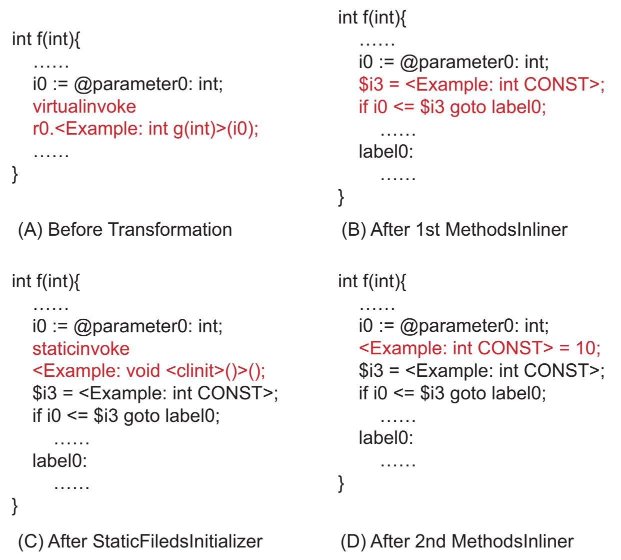


Fig. 3. Framework

Secondly, `StaticFieldsInitializer` adds methods `<clinit>()` of the classes, which are collected from the first step, to the head of program body. Lastly, `MethodsInliner` inlines those `<clinit>()` methods.

An example of a class called `Example` is shown in Figure 5. In the class, method `f` calls `g`, and `g` uses a static field of this class. With the last two transformations, the final predicate interpretation set is {`@parameter0` $\leq$ `10`}. Otherwise it will be {`@parameter0` $\leq$ `<Example: int CONST>`}, which is not as elegant as the former.



```
int f(int){
    ......
    i0 := @parameter0: int;
    virtualinvoke
    r0.<Example: int g(int)>(i0);
    ......
}
```

(A) Before Transformation

```
int f(int){
    ......
    i0 := @parameter0: int;
    $i3 = <Example: int CONST>;
    if i0 <= $i3 goto label0;
      ......
    label0:
      ......
}
```

(B) After 1st MethodsInliner

```
int f(int){
    ......
    i0 := @parameter0: int;
    staticinvoke
    <Example: void <clinit>()>();
    $i3 = <Example: int CONST>;
    if i0 <= $i3 goto label0;
      ......
    label0:
      ......
}
```

(C) After StaticFiledsInitializer

```
int f(int){
    ......
    i0 := @parameter0: int;
    <Example: int CONST> = 10;
    $i3 = <Example: int CONST>;
    if i0 <= $i3 goto label0;
      ......
    label0:
      ......
}
```

(D) After 2nd MethodsInliner

Fig. 4. Transformation

Comparing similar primitive approach in Soot, `MethodsInliner` have two significant advantages, (1) `MethodsInliner` uses a recursion algorithm to inline all methods that need to be inlined in the program body. (2) `MethodsInliner` takes polymorphism into account. For example, there is an invoke statement like `child.<Super: void func()>()` where `child` is an instance of a class whose super class is `Super`. In this situation, we firstly get the type of `child`, and redirect the method to the child class.

B. Data Flow Analysis

Data flow analysis is the main part of our tool. We define `PIAnalysis` extending `BackwardFlowAnalysis` of Soot. The input of this part is a `TrapUnitGraph` of the transformed Jimple code which adds edges from every trapped unit to the trap's handler unit.

`PIAnalysis` overrides several key methods, four of which are: `flowThrough()` corresponding to Data-Flow Equation, `merge()` corresponding to Merge($\wedge$), `entryInitialFlow()` corresponding to Boundary, `newInitialFlow()` corresponding to Initialize in Table 1.

The most important method is `flowThrough()`, in which we handle fifteen kinds of statements of Jimple [9]. Most of them need not do anything, because they use the identity transfer function. However, three kinds of them should be dealt with by ourselves using the transfer function (1) and (2). The first is branch statements including `IfStmt` and `SwitchStmt`. The second includes `IdentityStmt` and `AssignStmt`. And another one is `InvokeStmt`. In addition, induction is used for loops to reduce search spaces and improve the speed of analysis.

1) Branch Statement: Every `IfStmt` consists of a predicate and a goto statement. The predicate will be added to the set of pis according to the transfer function (1).

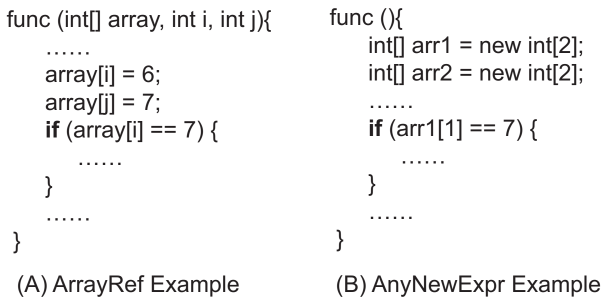
`SwitchStmts` have many branches. The tool creates a predicate for each value of switch statement. For instance, `switch(key){ case` v_1`: ···; case` v_2`: ···; ···}`, and {`key ==` v_1`, key ==` v_2`, ···`} will be added to the pi set.

2) Identity Statement and Assignment Statement: There is only one item in the two sides of the connector := in `IdentityStmt`. Given an identity statement which defines variable `i0` as the integer parameter, `i0 := @parameter0: int`, we use the transfer function (2) to replace an item with the other.

The most complex case is the `AssignStmt`. We must deal with many kinds of expressions on the two sides of =, such as `BinopExpr`, `ArrayRef`, `InvokeExpr`, etc. We must pay attention to two cases of them.

The first case is when the left side of = is an `ArrayRef`. In many earlier researches, arrays were considered as common objects whose fields were labeled with integer indexes. However, they ignored a big difference. Fields of a common object have fixed names, but if we consider an array as an object, the names of its fields will change at any time (e.g. modifying `array[i]` sometimes implies that `array[j]` is also changed if `i` equals to `j`). Figure 5(A) gives an example to explain how to deal with them. If we substitute `array[i]` with `6` according to the name, the condition expression will be `6 == 7`. However, if `i` equals to `j` during runtime, the condition expression will be `7 == 7`. Therefore, we keep the predicate instead of replacing the variable. For instance, `array[i]==7` will remain, and `i` can be considered as a reference variable whose value can be determined by some strategies (e.g. generating randomly, etc.) in further use.

The second case is when the right side of = is an `AnyNewExpr`. We keep these assignment statements to avoid confusion of different objects. Figure 5(B) presents an example. If we substitute `arr1` with `new int[2]`, we won't be able to distinguish `arr1` from `arr2`. In this example, the sequence {`arr1=new int[2], arr1[1]==7`} will remain.



```
func (int[] array, int i, int j){
    ……
    array[i] = 6;
    array[j] = 7;
    if (array[i] == 7) {
        ……
    }
    ……
}
```

(A) ArrayRef Example

```
func (){
    int[] arr1 = new int[2];
    int[] arr2 = new int[2];
    ……
    if (arr1[1] == 7) {
        ……
    }
    ……
}
```

(B) AnyNewExpr Example

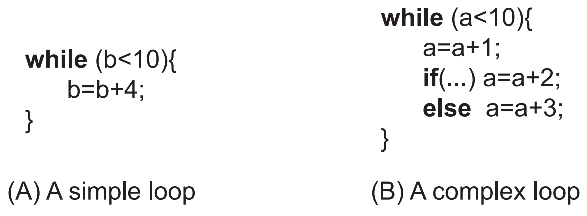
Fig. 5. Assignment Statement

3) Invoke Statement: Most `InvokeStmts` have been inlined into the program body except Java library methods in default case. In most situations, inlining all methods is impossible and unnecessary, and some of the remaining methods may affect the values of local variables. These methods will be put into the result set with predicates, because in static analysis we can't judge whether they will change their arguments or callers. For example, a sequence {`aList.clear()`, `aList.isEmpty() == 0`} will be kept. With the invoke statement `clear()`, `aList.isEmpty() == 0` will be considered as an input-independent pi. On the other hand, if we can confirm a method is insignificant, such as `println(Object)`, we will put them in a configuration file so that the method can be filtered by textbfFilter.

4) Loops: Loop is an important issue which will lead to producing too many similar pis like $a > 0$, $a{+}1 > 0$, $a{+}2 > 0$ and so on, and even infinite data-flow analysis. To avoid these problems, we define some reference variables to help present loop variables contained in loop conditions.

A predicate of a simple loop like the one shown in Figure 6(A) can be simply written as $b + 4 * i < 10$ where i is a reference variable. If a loop variable is modified in more than one places in the loop body (e.g. Figure 6(B)), we will define several reference variables for it. Assume the loop in Figure 6(B) is executed i times and the first branch of the if statement is executed j times, then the other branch must be executed $i - j$ times. The predicate for the loop will be written as $a + 2 * j + 3 * (i - j) + 1 * i < 10$ where i and j are reference variables that can't be determined in static analysis. If we nest the simple loop in Figure 6(A) to the end of the loop in Figure 6(B), the pis of the nested loops will be $\{b{+}4{*}m{*}n < 10, a{+}2{*}j{+}3{*}(n{-}j){+}1{*}n < 10\}$ where the simple loop executes m times, the outer loop executes n times, and the first branch of "if" executes j times.

In summary, induction for loop variables cannot only simplify the result set of pis, but also reflect the program structure more precisely.



```
while (b<10){
    b=b+4;
}
```

(A) A simple loop

```
while (a<10){
    a=a+1;
    if(...) a=a+2;
    else  a=a+3;
}
```

(B) A complex loop

Fig. 6. Loop

C. Filter

We obtain a set of pis after the data-flow analysis part. Some expressions in the pi set are redundant which should be removed. We implemented a filter to improve the results by filtering four kinds of expressions: (1) **Reduandante pis** defined as Opposite predicates or same predicates. For example, $a > b$ and $b < a$ are the same predicate in semantics, while $a > b$ and $a \leq b$ are opposite, however, reflecting the same program structure. (2) **Input-independent pis**, which are relations between constants, such as $2 == 3$ which must be false.(3) **Useless invoke statements**. For example, a method `set(Local)` may change a local variable, but a method `System.out.print()` is insignificant. The latter is "useless" and can be ignored. (4) **Useless assignment statements**. If a remaining assignment statement has nothing to do with pis in the result set, it can be deleted.

D. Output rule

It is important to make our tool work with different related tools, such as constraint solvers. In order to be convenient to transform results to other input format, we have established a grammar for the result expressions. Figure 7 shows the main part of the grammar where we define three result forms: (1) pi_stmt for pis. This is the main part of the final result set. (2) new_stmt for remaining assignment statements whose right side is `AnyNewExpr`. These expressions are used to distinguish different objects. It is explained in section 3.2.2. (3) $invoke\_stmt$ for remaining invoke statements which is described in section 3.2.3.

IV. Preliminary Experiment

We designed and conducted a preliminary experiment to verify our tool on two subject programs, `jtcas` and `ordset`, from SIR [3]. The common characteristics of the programs are a few of numerical or string inputs and simple data structure. It can help present the analysis results clearly.

The experiment was conducted in the default case which means Java library methods will not be inlined into the program body. A classic pi from the experiment is as follows

$$\begin{aligned}&staticinvoke\\ &<java.lang.Integer:\ int\ parseInt(java.lang.String)>\\ &\qquad(@parameter0:\ java.lang.String[][11]) <= 0\end{aligned}$$

where a Java library method, `parseInt()`, is kept. The first parameter of the analyzed program is a character string array.

```
stmt = pi_stmt | new_stmt | invoke_stmt

pi_stmt = expr rel_op expr;
rel_op = ">" | "<" | ">=" | "<=" | "==" | "!=";

new_stmt = local "=" any_new_expr;
any_new_expr = new_array_expr | new_multi_array_expr
                  | new_expr;

invoke_stmt = instance_invoke_expr | static_invoke_expr;
instance_invoke_expr = "instanceinvoke" immediate
                              ".<" method_signature ">(" expr_list ")";
static_invoke_expr = "staticinvoke"
                              "<" method_signature ">(" expr_list ")";

expr = immdiate |  arithmetic operation between expr

immediate = constant | local | param_ref | instance_field_ref
               | static_field_ref | array_ref | invoke_stmt;
param_ref = "@parameter" int_constant ":" type
instance_field_ref = immediate ".<" field_signature ">";
static_field_ref = "<" field_signature ">";
```

Fig. 7. Parts of the grammar for expressions

If the 11th element is parsed as an integer, the value will be greater than or equal to 0.

The `main` method in `jtcas` and a test driver method of `ordset` was analyzed. The `main` method in `jtcas` accepts a character string array as input. These command line arguments are transformed to integers and they are handled with complex logic to avoid traffic collision. The test driver method of `ordset` creates two ordered sets, and some operations, like union, adding elements, are completed on them.

The statistics results of the experiment are shown in Table 2. These pis have been divided into three types, each of which results in a markedly different effect on the domain boundary [10] in domain testing: Equalities($=$), Inequalities ($<$, $>$, $\leq$, $\geq$) and Non-equalities($\neq$).

TABLE II
EXPERIMENTAL RESULTS

statistic results	jtcas		ordset	
	#	percent	#	percent
Equalities	0	0%	7	2.8%
Inequalities	17	70.8%	189	75.0%
Non-equalities	7	29.2%	56	22.2%
Total	24	100%	252	100%
Redundant #	86	–	133	–
Preprocessing Time (ms)	330		610	
Analysis Time (ms)	60		430	

We can acquire some conclusions according to the statistic results as follows. (1) In-equalities ($<$, $>$, $\leq$, $\geq$) of a common program have a large proportion among all pis. The proportion is about 70%, even more. The final result pi set contains only a few equalities (=). (2) There are a large number of redundant pis before filtering. This result suggests that the filter part is necessary, and it can simplify the results greatly. (3) The time cost of PI analysis is collected to demonstrate the efficiency of our tool. It can produce all pis within several seconds for most common programs.

V. Related Work and Discussion

(pi) can be considered as a simplified version of path condition. pi focuses on the local structure information instead of the global. The compromise makes it more practical, especially in domain testing [10]. Jeng [4] integrated domain testing and data-flow testing which applied the pi to def-use chains in data-flow testing. Jeng and Weyuker [5] proposed a simplified domain testing strategy which was not limited to linear pis any longer. In addition, Zhao [11] presented a method to generate domain boundary for character strings.

PI analysis can also be done by forward data-flow analysis called symbolic analysis [1] which is regularly used in traditional optimizing compilers. The main idea of symbolic analysis is using parameters as reference variables to present local variables in the program body. However, if it is used in our approach, many resources will be wasted to analyze unconcerned variables, which may not be used in predicates later. Thus, our approach adopts backward data-flow analysis to get predicates firstly, and then analyze the related variables without any waste.

VI. Conclusion

In this paper, we introduce an approach for predicate interpretation analysis based on data-flow analysis. Besides, we implement a predicate interpretation analysis tool, which incorporates loop induction to reduce the search space and improve the speed of analysis. Further we conducted an experiment and the results show that our tool can produce pis of programs effectively and efficiently.

Acknowledgement

This work was partially supported by the National Basic Research Program of China (973 Program 2014CB340702), the National Natural Science Foundation of China (No. 61373013, 61170067), HPI and HPI Research School, Program B for Outstanding PhD Candidate of Nanjing University.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers–Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 2006.

[2] C. Cadar, P. Godefroid, and etc. Symbolic execution for software testing in practice–preliminary assessment. In *ICSE'11*, pages 1066–1071, 2011.

[3] G. R. Group et al. Software-artifact infrastructure repository (sir), 2009.

[4] B. Jeng. Toward an integration of data flow and domain testing. *Journal of Systems and Software*, 45(1):19–30, 1999.

[5] B. Jeng and E. J. Weyuker. A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology*, 3(3):254–270, 1994.

[6] G. Kildall. A unified approach to global program optimization. In *PLDI'73*, pages 194–206, 1973.

[7] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[8] R. Vallee-Rai, P. Co, E. Gagnon, and etc. Soot–a java bytecode optimization framework. In *CASCON'99*, pages 125–135, 1999.

[9] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, 1998.

[10] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, 1980.

[11] R. Zhao, M. R. Lyu, and Y. Min. Domain testing based on character string predicate. In *ATS'03*, pages 96–101, 2003.