JAAT: Java Alias Analysis Tool for Maintenance Activities

Fumiaki OHATA, Yusuke YAMANAKA,Kazuhiro KONDO and Katsuro INOUE*

November 29, 2003

Abstract

Alias analysis is a method for extracting sets of expressions which may possibly refer to the same memory locations during program execution. Although many researchers have already proposed analysis methods for the purpose of program optimization, difficulties still remain in applying such methods to practical software engineering tools in the sense of precision, extensibility and scalability.

Focusing mainly on a practical use for program maintenance activities such as program debugging and understanding, we propose an alias analysis method for object-oriented programs and discuss our implementation of JAAT.

Our proposed method employs a two-phase, on-demand, instance-based and extensible algorithm, in which intra-class analysis is done in Phase 1 for whole programs and libraries, and inter-class analysis is done in Phase 2 only for a user-demanded target. We can explore different algorithms by considering the trade-off between analysis precision and analysis cost by replacing Phase 1 (graph construction) or Phase 2 (graph traversal). JAAT can analyze large programs or libraries such as a JDK class library. Also, JAAT includes various features for program maintenance activities, such as GUI for displaying aliases, and an XML database for storing analysis information.

Contents Indicators: D: Software, D2: Software Engineering, D2.3: Coding Tools and Technologies (Object-oriented programming), D2.5: Testing and Debugging (Debugging aids), D2.7: Distributed, Maintenance, and Enhancement (Restructuring, reverse engineering, and reengineering)

Additional Keywords: Alias analysis, Maitenance, Java

1 Introduction

An *alias relation* between two expressions, e_0 and e_1, in a source program is a relation such that e_0 and e_1 may possibly refer to the same memory location during program execution. Alias relations are generated by various situations such as parameter passing, reference variables, and indirect reference with pointer variables. We say that e_0 is an *alias* of e_1 (and vice versa) when there is an alias relation between e_0 and e_1. Also, we call the set of expressions in

*The authors are with the Software Engineering Laboratory (C/O Katsuro Inoue), Department of Computer Science, Graduate School of Information Science and Technology, Osaka University, 1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, JAPAN. E-mail: inoue@ist.osaka-u.ac.jp.

which each element pair satisfies an alias relation, an *alias set*. *Alias analysis* is a method for extracting alias sets by static analysis. Moreover, alias analysis can be used for various purposes such as *compiler optimization* and *program slicing* [2,42].

Alias analysis was first proposed for traditional procedural languages such as C as part of the static analysis of pointer variables [14,21,25,35,37,43]. Concepts such as *class*, *inheritance*, *dynamic binding*, and *polymorphism* have been introduced through object-oriented (OO) languages such as C++ and JAVA [8,16,38]. Various alias analysis methods for OO programs have been devised [11,40]. This research focuses mainly on analysis algorithms as compiler optimization, but has not explored practicability and scalability of software engineering tools.

We are interested in developing a practical software engineering tool for the alias analysis targeting OO languages such as JAVA; however, implementation of already proposed approaches remains difficult as discussed by Hind et. al. [23].

We believe that two major issues exist, *scalability* and *the usage and approach of the analysis*:

Scalability: Since programs have become larger and class libraries associated with the developed programs tend to be huge and complex, the analysis should satisfy scalability in handling large programs within a reasonable time. However, many analysis methods produce poor results due to experimental implementations and more studies need to be conducted [23].

The usage and approach of the analysis: Most previous work focused mainly on compiler optimization and back-end for data-flow analysis as applications of the alias analysis. For such purposes, all alias relations in the program need to be extracted by the analysis. Nonetheless, alias analysis is useful as a software engineering tool for program maintenance activities [23].

For program debugging and understanding, not all alias relations are needed at one time; only user-requested relations are to be extracted quickly. Thus, we have to newly devise an on-demand, incremental analysis approach, which can be used effectively in an interactive environment. Note that in this paper, "alias analysis" means to extract a single set of expressions which are in alias relation to the user-specified expression, although a traditional meaning would be to extract all alias sets in a source program.

To resolve these issues, we propose an alias analysis method for OO programs, characterized as follows:

Two-phase and on-demand algorithm: We have developed a two-phase approach, in which intra-class analysis is done in Phase 1 for whole programs and libraries, and inter-class analysis is done in Phase 2 only for a user-demanded target. This two-phase approach greatly contributes to the overall performance of the analysis.

Flow-sensitive instance-based algorithm: Flow-sensitiveness of the analysis is an important factor in determining analysis precision and cost [20, 22, 34, 49]. We believe that a flow sensitive approach, rather than a flow insensitive one, is more useful for program maintenance activities, because of its focused results.

For OO programs in particular, an instance-based analysis for individual objects instantiated from a single class will preferably reduce the analysis result when compared to a class-based analysis (this is confirmed by our experiments), although the instance-based approach generally requires a high analysis cost. Furthermore, as a part of the instance-based analysis, we newly devise a method, called object context analysis, to remove surplus alias expressions in never-invoked instance methods.

In summary, we take a flow-sensitive instance-based analysis algorithm in this research to aim for a more focused result with a practically affordable analysis cost.

Extensible algorithm: The two-phase approach is also suitable for extending analysis algorithms since software components for each phase and sub-phase are easily replaceable. Therefore, a new analysis policy for a precision-cost compromise will be introduced.

We have implemented the proposed algorithm in the tool JAAT. JAAT considers scalability in the sense that it can analyze large programs with reasonable computation time. For example, the analysis time of 58,300 lines of Java programs with 364,721 lines of the JDK library was 30 seconds in Phase 1, and less than 1 millisecond in Phase 2. This result shows that the user can immediately get the resulting aliases on-demand for the user-specified analysis target after the preparation of Phase 1. Also, the result was fairly focused in the sense that for our sample programs, about 5 - 120 aliases were found due to the instance-based approach, which comprised 30 – 97% of the class-based approach. JAAT does not provide whole alias relations as the compiler optimization algorithm requires, but it makes the focused or scoped results useful for the program maintainers.

An additional feature of JAAT is that it can save internal syntactic and semantic information as an external XML database, and restore the information, in order to improve reusability of analysis results. Also, JAAT provides a useful Graphical User Interface (GUI), which shows the resulting aliases by using several visualizations and by supporting program maintenance activities.

In Section 2, we give a brief overview of alias analysis for OO programs. In Section 3 and Section 4, we propose an alias analysis method for OO programs. In Section 5, we discuss the algorithm complexity of our method. In Section 6, we introduce an implementation of the proposed method and evaluate its effectiveness using several sample programs. In Section 7, we discuss the evaluation results with respect to related works. In Section 8, we conclude our discussion with a few remarks and describe our future work.

2 Preliminary

Here, we show an example of aliases and their application, and classify existing alias analysis methods. Also, we discuss the problems of alias analysis for OO programs.

2.1 Example of Aliases

Alias analysis is useful for *program debugging* and *program understanding*. For an intuition of this, we present an example here.

```
class Employee {
  String name; int salary; Employee supervisor;
  Employee(String n, int s) {
    name = n;
    salary = s;
    supervisor = null;
  }
  void add_salary(int n) {
    salary += n;
  }
  void set_supervisor(Employee e) {
    supervisor = e;
  }
  void print() {
    System.out.println(name + ” Salary:” + salary);
  }
}
```

```
class Manager extends Employee {
  Manager(String n, int s) {
    super(n, s);
  }
  void manage(Employee e) {
    e.set_supervisor(this);
    e.add_salary(200);
  }
}
class Office {
  public static void main(String args[]) {
    Employee Emp = new Employee(”Emp”, 750);
    Manager Mng = new Manager(”Mng”, 750);
    Mng.manage(Emp);
    Emp.print();
    Mng.print();
  }
}
```

(a) Java source program

```
% java Office
Emp Salary: 950
Mng Salary: 750
```

(b) Program execution result with error

```
% java Office
Emp Salary: 750
Mng Salary: 950
```

(c) Program execution result without error

Figure 1: Simple debugging process of Java program with aliases

Fig.1(a) shows a sample JAVA program and Fig.1(b) shows its execution outputs. This program computes the salaries of employee `Emp` and manager `Mng`. The salary of the manager should be higher than that of the employee.

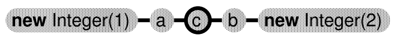
However, the program execution output is incorrect because a salary addition was made to `Emp`. When the user recognizes such a fault, he/she computes the aliases for reference variable `Emp` at line 32. In this paper, we call such a target expression of the alias analysis the *alias criterion* (or simply *criterion*), and it is specified by a tuple $<s, e>$, where s is a statement in the source program and e is an expression at s. In the figure, shadowed expressions represent the resulting aliases for $<s_{32}$, `Emp`$>$. $\boxed{\texttt{Emp}}$ at line 32 is the alias criterion and is also an alias itself. Therefore, it is boxed and shadowed. We can easily see around those shadowed expressions, and can identify a fault at the salary addition statement at line 24. By modifying the statement `e.add_salary(200)` to `add_salary(200)` at line 24, the program will compute an expected result as shown in Fig.1(c).

2.2 Alias Analysis

Alias analysis methods are roughly divided into two categories: *flow insensitive alias analysis (FI analysis)* and *flow sensitive alias analysis (FS analysis).*

```
1: Integer a, b, c;
2: a = new Integer(1);
3: b = new Integer(2);
4: c = b;
5: System.out.println(c);
6: c = a;
7: System.out.println(c);
```

(a) FI aliases



(b) Alias graph

Figure 2: Example of FI alias analyses

```
1: Integer a, b, c;
2: a = new Integer(1);
3: b = new Integer(2);
4: c = b;
5: System.out.println(c);
6: c = a;
7: System.out.println(c);
```

(a) FS aliases

Statement(s)	Reaching alias set($RA(s)$)
s_1	ϕ
s_2	ϕ
s_3	{{(s_2, `a`), (s_2, `new Integer(1)`)}}
s_4	{{(s_2, `a`), (s_2, `new Integer(1)`)}, {(s_3, `b`), (s_3, `new Integer(2)`)}}
s_5	{{(s_2, `a`), (s_2, `new Integer(1)`)}, {(s_4, `c`), (s_3, `b`), (s_4, `b`), (s_3, `new Integer(2)`)}}
s_6	{{(s_2, `a`), (s_2, `new Integer(1)`)}, {(s_4, `c`), (s_5, `c`), (s_3, `b`), (s_4, `b`), (s_3, `new Integer(2)`)}}
s_7	{{(s_6, `c`), (s_2, `a`), (s_6, `a`), (s_2, `new Integer(1)`)}, {(s_3, `b`), (s_4, `b`), (s_3, `new Integer(2)`)}}

(b) Reaching alias set (RAset)

Figure 3: Example of FS alias analyses

2.2.1 Flow Insensitive Alias Analysis (FI Analysis)

In FI analysis, we do not take into account the execution order of each statement in the source program [3, 21, 28, 35, 37, 49]. To compute FI aliases, an *alias graph*, as shown in Fig.2(b), is used. An alias graph is an undirected graph, in which each node represents an expression that refers to a particular memory location. Each edge represents a possible alias relation between two nodes, which occurs on both sides of an assignment statement and on the actual and formal parameters.

In Fig.2(a), when we specify $<s_7$, `c`$>$ as the alias criterion, we get aliases {`a`, `b`, `new Integer(1)`, `new Integer(2)`}, which are all reachable nodes from the criterion node in the alias graph.

Note that we have discussed the basic idea for FI analysis in this paper. The details are discussed in previous papers [3,21,28,35,37.49].

2.2.2 Flow Sensitive Alias Analysis (FS Analysis)

In FS analysis, we consider the execution order of statements [14, 26, 43]. To compute FS aliases, Landi et al. have introduced *a reaching alias set (RAset)* [26]. A RAset for statement s, denoted by $RA(s)$, is a collection of *alias sets*, which exists just before the execution of s. Each alias set is composed of sets of tuples (t, f) (t is a statement in the source program and f is an expression at t), meaning that each f at t in the set possibly refers to the same memory location. Fig.3(b) shows RAsets for each statement in Fig.3(a). In order to compute the aliases for $<s$, $e>$, we search $RA(s)$ for an alias set that contains e. At $RA(s_7)$ in Fig.3(b), since an alias set {(s_6, `c`), (s_2, `a`), (s_6, `a`), (s_2, `new Integer(1)`)} contains variable `c`, we get the result as shown in Fig.3(a).

Since FS analysis considers the execution order, it generally requires a larger amount of CPU time and memory space than FI analysis; however, FS analysis can extract more accurate alias relations than FI analysis. In Fig.2 and Fig.3, we can see the difference in the accuracy between the two methods. Previous work shows empirical comparisons of these analyses. In this paper, we focus on FS analysis for more accurate analysis results [20, 22, 49].

Note that our definition of the RAset is slightly different from that of Landi et. al. [26]. The latter tries to gather all expressions in alias relations existing in the entire target program at one time in order to perform compiler optimization, or back-end for data-flow analysis.

2.3 Alias Analysis for Object-Oriented Programs

Alias analysis methods for OO programs have been proposed as an extension of analysis methods for procedural programs [11, 40]; however, some issues still remain to be solved for the implementation of practical alias analysis tools for OO programs. Here, we consider three issues, as follows:

[a] Overall computation time for analysis

In order to implement a practical analysis tool, overall computation time is one of our main concerns. We have chosen a relatively expensive FS approach, where we must consider the execution order of statements to compute the RAset. When the target program contains loops or recursive method calls, it should be analyzed until the increase of the RAset settles down. In other words, when a $RA(s)$ for statement s changes during alias computation, we must re-compute the RAsets for all statements that are possibly affected by $RA(s)$. Thus, convergence and total computation time are important factors of the tool.

[b] Effective reuse of analysis results

The alias analysis tool should be used repeatedly with various alias criteria or with slightly distinct target programs. In such cases, we do not want to re-analyze the program. In a simple and straightforward FS approach, $RA(s)$ for statement s must be computed by analyzing all statements in the source program along the execution order. Therefore, when another statement t is modified, we might simply re-compute $RA(s)$ for each statement s in the source program even if $RA(s)$ is not affected by the modification of t. We are interested in finding an effective approach that re-computes only the RAsets affected by the modification.

[c] Improvement of analysis precision by separating each instance

In OO programs such as JAVA, each object has its own state and behavior even if they are instantiated from the same class. In sample JAVA program shown at Fig.4, we prefer to have three independent alias sets:

- {(s_1, `new Integer(1)`), (s_5, `x.get()`), (s_9, `id`), (s_{10}, `ref`), (s_{11}, `ref`), (s_{11}, `id`), (s_{15}, `id`)}
- {(s_2, `new Integer(2)`), (s_6, `y.get()`), (s_9, `id`), (s_{10}, `ref`), (s_{11}, `ref`), (s_{11}, `id`), (s_{15}, `id`)}
- {(s_4, `new Integer(3)`), (s_7, `z.get()`), (s_9, `id`), (s_{12}, `ref`), (s_{13}, `ref`), (s_{13}, `id`), (s_{15}, `id`)}

However, if we apply a simple analysis approach such that all objects instantiated from the same class share the alias information of their attributes and their calling-contexts, we get only one alias set which is the union of these 3 alias sets:

{(s_1, `new Integer(1)`), (s_5, `x.get()`), (s_2, `new Integer(2)`), (s_6, `y.get()`), (s_3, `null`), (s_4, `new Integer(3)`), (s_7, `z.get()`), (s_9, `id`), (s_{10}, `ref`), (s_{11}, `ref`), (s_{11}, `id`), (s_{12}, `ref`), (s_{13}, `ref`), (s_{13}, `id`), (s_{15}, `id`)}

In this case, many expressions are unwillingly in the same alias set.

```
1:  Ident x = new Ident(new Integer(1));
2:  Ident y = new Ident(new Integer(2));
3:  Ident z = new Ident(null);
4:  z.set(new Integer(3));
5:  ... = x.get();
6:  ... = y.get();
7:  ... = z.get();
```

```
8:  class Ident {
9:     private Integer id;
10:    public Ident(Integer ref)
11:       { id = ref; }
12:    public void set(Integer ref)
13:       { id = ref; }
14:    public Integer get()
15:       { return id; }
16: }
```

Figure 4: Example of aliases across instances

In order to increase the analysis precision, we will separately hold the alias information for each attribute of each object instance, although this approach generally requires more analysis cost. However, we will devise an efficient approach to resolve this.

These issues are also major concerns for traditional programming languages, and they have been studied in various research [10,20,43,48]. Moreover, these issues are even more critical problems of alias analysis for OO languages such as JAVA because of the nature of OO languages.

3 Analysis Overview

In this section, we provide an overview of our proposed method. In the following, we explain three analysis policies and define a new term, object context. Also, we discuss many issues of the alias analysis for OO programs. The details of our proposed method will be shown in the next section.

3.1 Approach

Here, we divide alias relations into two categories, *intra-class alias relations* and *inter-class alias relations*. Inrta-class alias relations do not depend on their usage contexts. Inter-class alias relations are obtained by analyzing expressions with method invocations and reference variables over classes.

For example, in Fig.4, $\{(s_9$, `id`), (s_{10}, `ref`), (s_{11}, `ref`), (s_{11}, `id`)$\}$, $\{(s_9$, `id`), (s_{12}, `ref`), (s_{13}, `ref`), (s_{13}, `id`)$\}$ and $\{(s_9$, `id`), (s_{15}, `id`)$\}$ are intra-class alias relations, and $\{(s_1$, `new Integer(1)`), (s_{10}, `ref`)$\}$, $\{(s_5$, `x.get()`), (s_{15}, `id`)$\}$, $\{(s_2$, `new Integer(2)`), (s_{10}, `ref`)$\}$, $\{(s_6$, `y.get()`), (s_{15}, `id`)$\}$, $\{(s_3$, `null`), (s_{10}, `ref`)$\}$ and $\{(s_4$, `new Integer(3)`), (s_{12}, `ref`)$\}$ are inter-class alias relations.

Here, we will adopt the following analysis policies:

Policy 1: Compute intra-class alias relations in advance.

Policy 2: Compute inter-class alias relations on-demand.

Utilizing these two policies, the modularity and independence of the analysis will be established. This is particularly important in OO programming, since we usually use large class libraries in addition to user-developed classes. The analysis cost of these class libraries is generally large. Thus, modularizing the analyses of class libraries and user-developed classes is essential.

Note that mixing inter-class alias relations reduces precision. In the case of Fig.4, mixing all alias expressions will unwillingly generate a large and useless alias set. In order to resolve this problem, we use the following policy:

Policy 3: Compute inter-class alias relations based on the individual invocations and references of instance methods and attributes (we call this *instance-based analysis*).

The instance-based analysis of OO programs can be considered to be an extension of the context-sensitive analysis of procedural programs.

3.2 Object Context

To further improve the analysis precision of the instance-based analysis of OO programs, we introduce a notion of object context.

Consider an alias set $\mathcal{A}$, which contains expressions referring to object instances. Some instance methods of these objects are invoked directly or indirectly from the expressions in $\mathcal{A}$, and some are never invoked from the context of $\mathcal{A}$. We delete unnecessary alias expressions which appear in the body of never-called instance methods to improve analysis precision.

The *object context* for alias set $\mathcal{A}$, denoted by $OC(\mathcal{A})$, is a set of instance methods that are in instance objects pointed to by expressions in $\mathcal{A}$, and that may be invoked from some expression associated with expressions in $\mathcal{A}$.

The concept of object context is inspired by a static analysis for virtual method resolution [31, 39]. By statically analyzing virtual method resolution using *rapid type analysis (RTA)* via a pointer analysis or a data-flow analysis, compilers can optimize the method invocation process. The object context analysis collects method invocation information for the specific object, by applying static virtual method resolution techniques to all the method invocations related to the object.

```
$OC(\mathcal{A}) := \phi$
foreach $(s, e)$ in $\mathcal{A}$ do
  if an invocation of instance method $m$ appears in expressions such as $e.m(\ldots)$ then
    if $m \notin OC(\mathcal{A})$ then
      $OC(\mathcal{A}) := OC(\mathcal{A}) \cup \{m\}$
    endif
  endif
end
repeat
  foreach $m$ in $OC(\mathcal{A})$ do
    if an invocation of an internal instance method $n$ appears in $m$'s body
        such as $this.n(\ldots)$ (i.e., $m$ possibly calls $n$) then
      if $n \notin OC(\mathcal{A})$ then
        $OC(\mathcal{A}) := OC(\mathcal{A}) \cup \{n\}$
      endif
    endif
  end
until $OC(\mathcal{A})$ is unchanged
```

Figure 5: Algorithm for object context analysis

The object context is formally obtained by the algorithm shown in Fig.5. In the first iteration, all instance method invocations directly associated with expressions in $\mathcal{A}$ are collected [1]. In the second loop, instance methods indirectly invoked in the same class are collected.

Fig.6 shows an example of the object context. Assume that $\mathcal{A}$ is the aliases for `a` at line 22 such that $\{(s_{20},$ `a`$),$ $(s_{20},$ `new Calc()`$),$ $(s_{22},$ `a`$)\}$. Now we know that $\mathcal{A}$ is a Calc type, and possible instance method invocations are `new Calc()` at line 20 and `a.inc()` at line 22. Thus, `Calc::Calc()` and `Calc::inc()` are included in $OC(\mathcal{A})$. Since these two methods have no further invocations of other instance methods in a `Calc` class, we finally know that $OC(\mathcal{A})$ is $\{$`Calc::Calc()`, `Calc::inc()`$\}$.

Also, if we assume that $\mathcal{A}$ is an alias set for `b` at line 21, then $OC(\mathcal{A})$ is $\{$`Calc::Calc()`, `Calc::add()`, `Calc::result()`$\}$.

When we compute the aliases for $e.i$ such that e is in $\mathcal{A}$, we can limit the candidate methods to be considered further by using $OC(\mathcal{A})$. In other words, we can exclude the instance methods that can not be invoked in the objects referred to by expressions in $\mathcal{A}$. In the case that we can not specify a unique $\mathcal{A}$'s type, we might have more than one method; *method overriding* would cause such a situation.

As in many other analysis methods, we can also consider two variants of the object context, *flow insensitive object context (FIOC)* and *flow sensitive object context (FSOC)*. The former takes into account the method invocation order, whereas the latter does not. FSOC extracts more accurate alias information than FIOC; however, FSOC requires

[1] Only the expressions in $\mathcal{A}$ are focused.

```
public class Calc {
  Integer i;
  public Calc() {
    i = new Integer(0);
  }
  public void inc() {
    i = new Integer(i.intValue() + 1);
  }
  public void add(int c) {
    i = new Integer(i.intValue() + c);
  }
  public Integer result() {
    return(i);
  }
}
```

```
class Test {
  Calc a, b;
  Integer c;
  Test() {
    a = new Calc();
    b = new Calc();
    a.inc();
    b.add(1);
    c = b.result();
  }
}
```

Figure 6: Example program for object context analysis

more analysis cost than FIOC. In our system, we use the FIOC approach, which is computed by the above mentioned algorithm. We will discuss FSOC in Section 7.

Note that our object context is an orthogonal notion of the instance-based analysis. In our system, after resolving the object context, we will perform the instance-based analysis as part of the inter-class analysis.

3.3 Other Issues of OO Program Analysis

OO languages like Java contain more features than traditional procedural languages. We use the following approaches for each feature.

Inheritance: The inheritance concept causes other features such as method overriding and dynamic binding. In addition, we must take virtual method invocation mechanisms into account, so that inter-class alias analysis algorithms must consider the inheritance.

Method overriding and dynamic binding: Method overriding might generate two or more methods that have the same signature in a class hierarchy. Since the determination of the invoking method depends on the reference-type of the object that receives a message, static identification of the actually invoked method is difficult. This difficulty stems from an undecidable type of receiver object without program execution; however, with alias analysis we can infer such a type more accurately.

For example, when we identify the invoking method of expression $a.b()$, we can use the alias information of a in order to infer the reference-types of the instances that might be referred to by a.

Typecasts: Our alias analysis is not affected by typecast operations, since we determine the reference-type of an expression e using class instance creation expressions in e's aliases.

Constructors: We deal with constructors as ordinary methods except that we should associate method invocation **this**(...) or **super**(...) with its corresponding constructor in this class or super class.

Static fields and methods: Our alias algorithm focuses on only user-requested alias relations in a specific scope or object. For a given alias criterion, we analyze only classes whose methods are probably invoked from the class with the criterion, or where instance variables are probably accessed from the class. Therefore, if expressions which access static fields or methods are in the user-requested scope or object, they are considered. On the other hand, if expressions which access to static fields or methods are outside the scope or object, they are ignored.

Our approach might be insufficient for collecting whole and precise alias information at one time; however, it is sufficient and effective for interactively computing aliases under program maintenance activities.

4 Details of Analysis

Based on the above mentioned Policy 1 – 3 and the object context, we propose the following two-phase approach.

Phase 1: Intra-class analysis for all source programs are composed of the following two sub-phases:

(a) Construction of AFG (defined later) by analyzing inside each method.

(b) Construction of MFG (defined later) by analyzing methods in each class.

Phase 2: Inter-class analysis for a specified alias criterion, i.e., computation of the aliases by traversing AFG and MFG along with the object context.

In this section, we describe the details of this approach.

4.1 Phase 1: Construction of AFG and MFG

4.1.1 Phase 1(a): Construction of AFG

An *alias flow graph (AFG)* is an undirected graph which shows FS alias relations inside a single method. A node represents either

- an expression that refers to an object (e.g., a reference variable, an instance creation expression, or a method invocation) or

- a parameter to/from a method or an instance.

A node representing an expression referring to an object is called an *AFG normal node.* A node with regard to methods or classes is called an *AFG special node.* Each special node is created according to the following rules:

- For each method invocation such as $a.m(a_1, a_2, \ldots, a_n)$ in AFG, we add an *method invocation (MI)* node. Also, for each actual parameter $a_1, a_2, \ldots, a_n$, we add an *actual alias in (AA-in)* node.
- For each definition of instance method such as $m(f_1, f_2, \ldots, f_n)$ of class C, the following special nodes are added in the AFG.
  - For each formal parameter, a *formal alias in (FA-in)* node. [2]
  - For each attribute of C, a *instance alias in (IA-in)* and a *instance alias out (IA-out)* node.
  - For each return expression, a *method alias out (MA-out)* node.

Except for MI, these AFG special nodes are added only when they are reference-type expressions.

An edge in AFG denotes an alias relation immediately determined inside each method. Alias relations created by assignment statements, variable definitions and their uses (def-use relations), and assignments of parameters to/from special nodes are called *direct alias relations.* Direct alias relations are easily obtained by RAset-based FS may-alias analysis inside methods [26]. Also, a path formed with more than one edge is called an *indirect alias relation.*

Note that an expression specifying an attribute b (or a method $b()$) associated with an instance a is denoted by $a.b$ (or $a.b()$) in JAVA[3]. In such a case, we say that the node for a in AFG is a *parent* of the node for b, and the node for b is called a *child* of the node for a (although we show no explicit edge for this relation in the figures of AFG). A parent-child relationship is used for the alias computation in Phase 2. Also, a parent-child relationship between an MI node and its corresponding AA-in nodes is created at this phase.

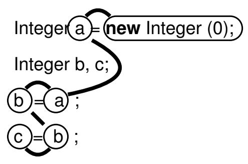
Fig.7 shows a small JAVA program and its AFG. Nodes in AFG are shown as circles with expressions inside, and the edges are denoted with solid lines. Other strings out of those nodes (e.g., `Integer`, `=`, `Integer b, c;`) are comments used to identify the occurrences of expressions and to help the reader imagine the original source text. In Fig.7(b), since (s_1, `new Integer(0)`) is assigned to (s_1, `a`) in the source program, we can see that the node for (s_1, `new Integer(0)`) is connected to the node for (s_1, `a`) with an edge. This edge represents a direct alias relation.

[2]There is no *actual alias out (AA-out)* or *formal alias out (FA-out)* node in JAVA. The former is alias passed by actual parameter to caller, and the latter is alias passed by formal parameter to caller. This is because JAVA only uses a passed-by-value mechanism. We discuss the cases of AA-out and FA-out in our technical report [30].

[3]When class C has an instance attribute b, we can refer to b in the form of both b and **this**.b in the method of C. In this case, we consider **this**.b as simply b.

```
1:    Integer a = new Integer(0);
2:    Integer b, c;
3:    b = a;
4:    c = b;
```

(a) Source program



(b) AFG

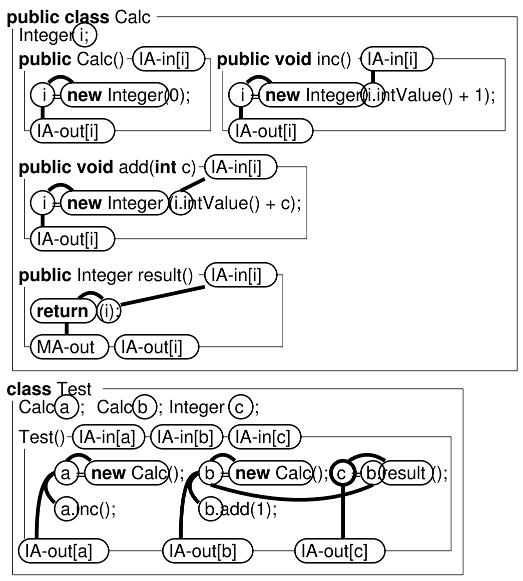
Figure 7: Example of AFG

Fig.8 is a Java program with two class definitions and its AFG. Variable `i` appearing at each right-hand side expression (line 7 and 10) is a reference-type instance variable. Thus, AFG special nodes IA-in[`i`] and IA-out[`i`] are employed for each method in class `Calc`. Also, the expression for the return value, `return(i)`, at line 13 is a reference to an object. Therefore, an AFG special node MA-out is created for method `Calc::result()`.

```
1:  public class Calc {
2:     Integer i;
3:     public Calc() {
4:        i = new Integer(0);
5:     }
6:     public void inc() {
7:        i = new Integer(i.intValue() + 1);
8:     }
9:     public void add(int c) {
10:       i = new Integer(i.intValue() + c);
11:    }
12:    public Integer result() {
13:       return(i);
14:    }
15: }
```

```
16: class Test {
17:    Calc a, b;
18:    Integer c;
19:    Test() {
20:       a = new Calc();
21:       b = new Calc();
22:       a.inc();
23:       b.add(1);
24:       c = b.result();
25:    }
26: }
```

(a) Source program



public class Calc
Integer i;
public Calc() IA-in[i]
i = new Integer(0);
IA-out[i]
public void inc() IA-in[i]
i = new Integer(i.intValue() + 1);
IA-out[i]
public void add(int c) IA-in[i]
i = new Integer(i.intValue() + c);
IA-out[i]
public Integer result() IA-in[i]
return (i);
MA-out
IA-out[i]
class Test
Calc a; Calc b; Integer c;
Test() IA-in[a] IA-in[b] IA-in[c]
a = new Calc(); b = new Calc(); c = b.result();
a.inc();
b.add(1);
IA-out[a]
IA-out[b]
IA-out[c]

(b) AFG

Figure 8: Example of AFG with special nodes

`b.result()` at line 24 is represented in AFG with a parent node `b` and a child node `result()`.

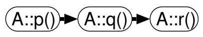
4.1.2 Phase 1(b): Construction of MFG

A *method flow graph (MFG)* is a directed graph, which represents the caller-callee relations of methods in a single class[4]. An *MFG node* denotes the definition of each method, and when method A possibly calls method B, an *MFG edge* is drawn from the node for A to the node for B.

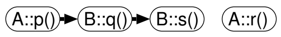
Fig.9 shows a sample program and its MFG. Method `p()` is not defined in class `B`, and method `A::p()` is executed when `p()` is activated on `B`'s object. In this case, method call to `q()` appearing in `A::p()` causes activation of `B::q()`, not `A::q()`. Thus, the resulting MFG for class `B` is as shown in Fig.9(b).

```
class A {
   void p() { q(); }
   void q() { r(); }
   void r() { }
}
```

```
class B extends A {
   void q() { s(); }
   void s() { }
}
```



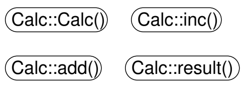
(a) class `A`



(b) Class `B`

Figure 9: Example of MFG

Fig.10 shows the MFG for the class `Calc` and the class `Test` shown in Fig.8(a). Since neither class has intra-class method calls, no MFG edge exists.



(a) Class `Calc`

(b) Class `Test`

Figure 10: MFG for Fig.8(a)

4.2 Phase 2: Alias Computation Using AFG and MFG

4.2.1 Traversal of AFG

Using AFG and MFG, we compute aliases $\mathcal{A}(e)$ for an alias criterion e, which is a reference-type expression. e itself is also an element of $\mathcal{A}(e)$. The nodes in AFG are visited beyond the class boundary using MFG information.

We show the overall algorithm of Phase 2 in Fig.11. For the readability of the algorithm, we use intuitive descriptions. A more formal definition is presented in our technical report [30]. Also, for the simplicity of the description, we

[4]MFG corresponds to a caller-callee graph (call graph) in procedural languages.

use the same symbol for an expression in the program text and for its corresponding AFG node.

The following is an overview of the traversal algorithm.

1. When we compute the aliases for e with a parent p, first we compute p's aliases $\mathcal{A}(p)$, and then we collect information about $\mathcal{A}(p)$, such as

   - types for $\mathcal{A}(p)$ and
   - OC($\mathcal{A}(p)$).

   After computing these, we compute e's aliases. When computing the OC($\mathcal{A}(p)$), we use MFG at the second loop in the OC computation algorithm (see Section 3). There are many cases where we can not compute the aliases for e without $\mathcal{A}(p)$ and their types. If we do not know p's aliases, we would have to consider that p could refer to all the objects instantiated from the classes derived from p, so that e's alias result would be enlarged.

   We have named this the *parent-first-child-last* approach.

2. When we reach an MI, MA-out, an FA-in or an AA-in node during AFG traversal, using MFG we determine the callee or caller method using the object context, and then we traverse from the corresponding MA-out, MI, AA-in or FA-in node, respectively.

For the programs with recursive method calls, AFG traversal terminates as discussed in the following section.

As an example of Phase 2, we show an alias computation process for <s_{24}, `c`> (boxed with a bold line) in Fig.12, which shows the same program as Fig.8(a).

1. Start AFG traversal from alias criterion (s_{24},`c`), and immediately reach method invocation (MI node) (s_{24}, `result()`) (Fig.13(a)).

2. Since (s_{24}, `result()`) has a parent node (s_{24}, `b`) (denoted by B), first compute (s_{24}, `b`)'s aliases to specify the object related to (s_{24}, `result()`)'s aliases.

   (a) Compute (s_{24}, `b`)'s aliases $\mathcal{A}(B)$. The result is {(s_{21}, `new Integer()`), (s_{21}, `b`), (s_{23}, `b`), (s_{24}, `b`)}.

   (b) Compute the types of $\mathcal{A}(B)$ using class instance creation expressions included in $\mathcal{A}(B)$. In this case, the type is determined to be `Calc`.

   (c) Compute OC($\mathcal{A}(B)$). The result is {`Calc::Calc()`, `Calc::add(int c)`, `Calc::result()`}. We know that `Calc::inc()` is never invoked in the context of b's aliases.

```
Input:
        an AFG $G = (V, E)$ for target classes;
        an alias criterion $e \in V$;
Output:
        alias set $\mathcal{A}(e)$;
Variables:
        $\mathcal{A}(e)$; /* set for the resulting alias nodes */
        $S$; /* set for the nodes not yet checked */
        $c$; /* a node in $V$. */

begin
  $\mathcal{A}(e) := \phi$;
  $S$ := all reachable nodes from $e$;
  while $S \neq \phi$ do begin
    remove a node $c$ from $S$ and add $c$ to $\mathcal{A}(e)$;
    case $c$ of
    reference to a variable: begin
      if $c$ has a parent $p$ such as $p.c$ then
        determine possibly-called methods $m$ using the context of $p$;
          /* $p$'s context is computed by $\mathcal{A}(p)$, $\mathcal{A}(p)$'s type and OC($\mathcal{A}(p)$) */
        add to $S$ $m$'s entry nodes for $c$; /* IA-in node, where $c$ may be referred in $m$ */
        add to $S$ $m$'s exit nodes for $c$; /* IA-out node, where $c$ may be defined in $m$ */
        add to $S$ nodes $c'$ such that $q.c'$ where $q \in \mathcal{A}(p)$ and $c$ and $c'$ are the same variables;
          /* $c$ may be accesses through different parent variables */
      endif; /* else if no parent, do nothing */
    end;
    method invocation: begin /* MI node */
      if $c$ has a parent $p$ such as $p.c()$ then
        determine possibly-called methods $X::c()$ in the context of $p$; /* by OC($\mathcal{A}(p)$) */
      else /* no parent */
        determine possibly-called methods this::c in the context of this; /* by OC($\mathcal{A}(this)$) */
      endif;
      add to $S$ $c$'s return expression node; /* traverse from the callee tail, MA-out */
    end;
    entry (or exit) node of instance variable: begin /* IA-in or IA-out node */
      determine possibly-called methods $m$ in the context of this; /* by OC($\mathcal{A}(this)$) */
      add to $S$ $m$'s exit (or entry) node of the instance variable; /* IA-out or IA-in node */
    end;
    actual parameter: begin /* AA-in */
      determine callee method $m$ with $c$ as its parameter;
      add to $S$ $m$'s formal parameter node corresponding to $c$;
    end;
    formal parameter: begin /* FA-in */
      determine caller method $m$ in the context of this; /* by OC($\mathcal{A}(this)$) */
      add to $S$ $m$'s actual parameter node;
    end;
    end; /* case */
  end; /* while */
end;
```

Figure 11: Overall algorithm in Phase 2

3. Since the alias computation for (s_{24}, `b`) indicates that it refers to the objects that are instantiated from class `Calc`, traverse AFG from the return expression (MA-out node) in `Calc::result()` (Fig.13(b)).

   (a) Reach the entry of an instance variable `i` (IA-in[`i`]) in `Calc::result()`.

   (b) Traverse from the exits of instance variable `i` (IA-out[`i`]) at methods `Calc::Calc()`, `Calc::add(int c)` and `Calc::result()`.

The resulting aliases are the masked expressions in Fig.12. Since OC($\mathcal{A}(B)$) does not contain `Calc::inc()`, expressions in `Calc::inc()` are excluded from the candidates for (s_{24}, `c`)'s aliases.

Together with Phase 1, the overall analysis algorithm establishes a FS, may-alias, instance-based, and FIOC approach.

```
1:  public class Calc {
2:     Integer i;
3:     public Calc() {
4:        i = new Integer(0);
5:     }
6:     public void inc() {
7:        i = new Integer(i.intValue() + 1);
8:     }
9:     public void add(int c) {
10:       i = new Integer(i.intValue() + c);
11:    }
12:    public Integer result() {
13:       return(i);
14:    }
15: }
```

```
16: class Test {
17:    Calc a, b;
18:    Integer c;
19:    Test() {
20:       a = new Calc();
21:       b = new Calc();
22:       a.inc();
23:       b.add(1);
24:       c = b.result();
25:    }
26: }
```

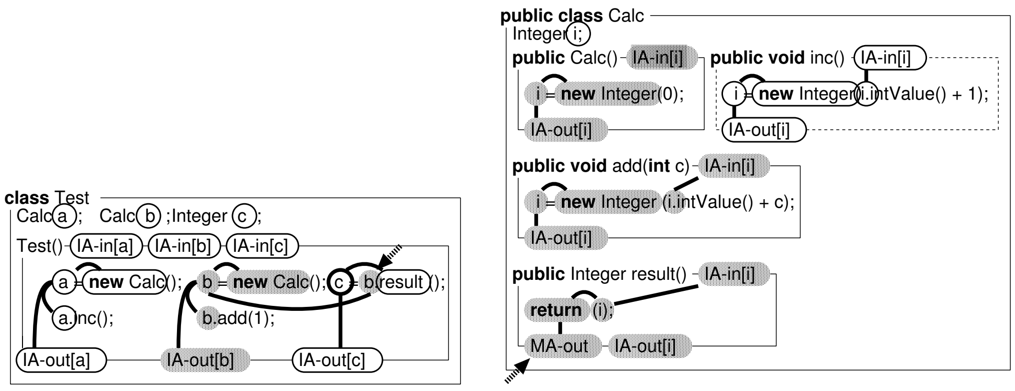
Figure 12: Aliases for <s_{24}, `c`> (masked expressions are aliases)

4.2.2 Algorithm Termination

For the programs without recursive structures, the algorithm shown in Fig.11 always terminates for a finite-size target program.

For the programs with recursive structures, we replace the statement at line 14 of Fig.11, 'remove a node c from S and add c to $\mathcal{A}(e)$', and its successive case statements, with the following statements.

- For each node $c \in S$, we add c to RNlist(e) (RNlist(e) represents *reached node list (RNlist)* for e).
  - If $c \in$ RNlist(e), we remove c from S *without* dding c to $\mathcal{A}(e)$. Also, the successive whole case statements are skipped.


class Test
Calc a; Calc b ;Integer c;
Test() IA-in[a] IA-in[b] IA-in[c]
a = new Calc(); b = new Calc(); c = b.result();
a.inc();
b.add(1);
IA-out[a] IA-out[b] IA-out[c]
public class Calc
Integer i;
public Calc() IA-in[i]
i = new Integer(0);
IA-out[i]
public void inc() IA-in[i]
i = new Integer(i.intValue() + 1);
IA-out[i]
public void add(int c) IA-in[i]
i = new Integer (i.intValue() + c);
IA-out[i]
public Integer result() IA-in[i]
return (i);
MA-out IA-out[i]

(a) <s_{24}, `b`>　　　(b) <s_{24}, `result()`>

Figure 13: Aliases in AFG for Fig.12 (masked expressions are aliases)

   – If $c \notin$ RNlist(e), we remove c from S with adding c to $\mathcal{A}(e)$. The case statement is executed as Fig.11.

- An RNlist is created for each alias set. For example, when $x.y$ is an alias criterion, RNlist(x) and RNlist(y) are created.

- Suppose that $x.y.z$ is an alias criterion and we have reached node c in the alias computation for x. When we check if $c \in$ RNlist(x), we should also check if $c \in$ RNlist(y) and check if $c \in$ RNlist(z), respectively.

Since the size of each RNlist is less than E (total number of expressions in the target program) and the number of the RNlists is less than k (maximum length of the parent-child chain), we can prevent infinite recursive structures or endless loop statements by creating RNlists so that the AFG traversal always terminates.

5 Complexity Analysis

In this section, we discuss the complexity of each phase of the algorithm.

TABLE 1 shows the meaning of the symbols used in the complexity expressions here.

- Phase 1(a): Construction of AFG

    When the target program has loop statements, we must analyze expressions at most E^2 times. In this case, expressions mean reference-type expressions, such as variables, parameters, and non-void method invocations. In order to analyze each expression, two or three set operations are needed; we remove the alias relations with regard to defined variables from RAset, and add a new alias relation

Table 1: meaning of symbols used for complexity expressions

Symbol	Description
A	Maximum number of attributes in each class (inherited attributes are also counted)
M	Maximum number of methods in each class (inherited methods are also counted)
L	Maximum number of sum of local variables and parameters in a method
C	Total number of classes in the target program
E	Total number of expressions in the target program
k	Maximum length of parent-child chains (when a chain forms a recursive loop, we do not count the previously visited expressions)

between the defined and the referred variables to the RAset, and so on. Since the set operation cost is proportional to the number of elements in the target set, the time complexity of the set operation is covered by $O(A+L)$. Thus, in the worst case, the time complexity is $O((A+L)\cdot E^2)$. The number of AFG nodes is $O(E)$, and the number of AFG edges is $O(E^2)$. The space complexity is $O(E^2)$ in the worst case. In our experimentation, however, both the time consumption and the space usage grew in near liner order.

- Phase 1(b): Construction of MFG

  One MFG component is constructed for each class. Since we must check each expression once in order to find method calls, the time complexity is $O(E)$. Since the number of methods in a class is less than M, the number of MFG nodes is $O(C\cdot M)$, and the number of MFG edges is $O(C\cdot M^2)$. Thus, the space complexity is $O(C\cdot M^2)$ in the worst case. Also, the actual time and space grew in near liner order in our experimentation.

- Phase 2: Alias computation using AFG and MFG

  Since we use the instance-based approach, we must also compute the aliases for parent nodes, recursively. In the worst case, the time complexity and the space complexity are both $O(E^k)$; however, such a case is quite rare. In our experimentation, k's values were 2 or 3 on average.

6 A Java Alias Analysis Tool (JAAT)

We have implemented the proposed method in the tool Java *Alias Analysis Tool (JAAT).* Using JAAT, we have analyzed several programs and obtained various data.

6.1 Overview of JAAT

JAAT consists of three subsystems, the *analysis subsystem*, the *XML database subsystem*, and the *user interface (UI) subsystem*. Fig.14 shows the structure of JAAT. We will present an overview of each subsystem.

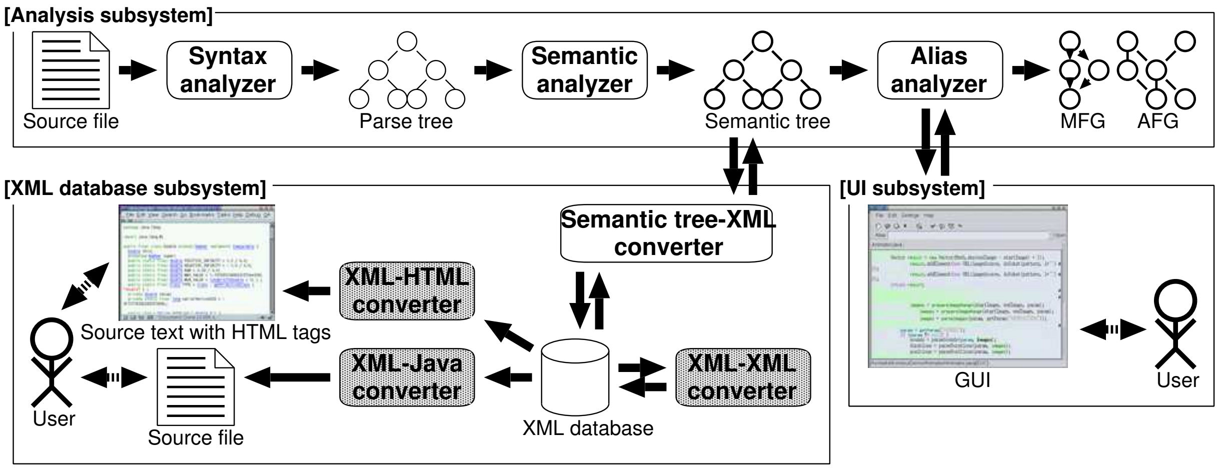


Figure 14: Architecture of JAAT

Analysis subsystem: The analysis subsystem consists of three components, the syntax analyzer, the semantic analyzer, and the alias analyzer. The *syntax analyzer* analyzes JAVA source files and generates syntactic trees[5]. The *semantic analyzer* proceeds with a semantic analysis that creates symbol tables and extracts declare-refer relations among identifiers and generates semantic trees. The *alias analyzer* generates MFGs and AFGs at Phase 1, and computes the aliases for the alias criterion specified by the user's request at Phase 2. The alias analyzer returns the resulting aliases to the UI subsystem.

XML database subsystem: Generated trees and graphs are proprietary data structures, and are placed into the memory space of JAAT, as are many other analysis tools [5,44]. Since the translation from a source program to the corresponding semantic tree is a fairly time-consuming process, we do not want to discard analysis results from the analysis sessions. Thus, we build a database for semantic trees. This feature improves the reusability of the analysis results along with the AFG and MFG approaches.

We use an *extensible markup language (XML)* database that holds semantic tree information [15]. The *XML converters* converts semantic trees to XML documents and vice versa. [6]

[5]The syntax analyzer is generated by ANTLR [4].

[6]These converters use libxml as an XML parser and Xalan-C++ as an *extensible stylesheet language transformations (XSLT)* processor [45–47].

UI subsystem: The UI subsystem[7] has two main functions, editing programs and visualizing the resulting aliases. Examples will be shown in Section 6.2.5.

6.2 Evaluation

In order to explore the applicability of JAAT, we have applied it to various sample programs. TABLE 2 shows features of the sample programs. Note that we must analyze not only these sample programs but also all related classes in JDK for inter-class alias analysis. For example, `TextEditor` is composed of one file with 1,125 lines. `TextEditor` directly and indirectly uses the classes in JDK, which are in 878 files with a total of 351,890 lines (99.7% of the overall total lines). This data shows that a heavy effort must be put on the analysis for the related classes in JDK.

Table 2: characteristics of analyzed sample programs

Programs	Sample Program		Related Classes in JDK	
	Number of Files	Number of Lines	Number of Files	Number of Lines
TextEditor	1(0.1%)	1125(0.3%)	878(99.9%)	351,890(99.7%)
JLex (Parser Generator)	1(0.4%)	7,835(6.9%)	275(99.6%)	105,234(93.1%)
java_cup (Parser Generator)	35(11.3%)	10,610(9.1%)	274(88.7%)	105,598(90.9%)
JFlex (Parser Generator)	40(4.3%)	13,029(3.6%)	882(95.7%)	353,067(96.4%)
WeirdX (X server)	47(5.0%)	19,701(5.2%)	892(95.0%)	356,217(94.8%)
ANTLR (Parser Generator [4])	129(31.6%)	25,283(19.3%)	279(68.4%)	105,483(80.7%)
Ant (Build Tool)	98(24.0%)	26,428(18.8%)	310(76.0%)	114,262(81.2%)
DynamicJava (JAVA Interpreter)	242(21.1%)	58,300(13.8%)	903(78.9%)	364,721(86.2%)

6.2.1 Computation Time of Phase 1(a)

Our modularized analysis is effective in that we only have to re-analyze modified parts of the programs when small parts of the program are modified. On the other hand, there are several FS analysis algorithms already proposed [14,26,43]. Those algorithms mainly focus on language-specific problems such as pointers to the stack, and they do not concern the separation of analysis results for each module. Therefore, if we would employ those algorithms, the overall program have to be re-analyzed.

It should be noted that user-written programs are often modified but their related classes in JDK are seldom modified.

TABLE 3(a) shows AFG construction time for sample programs and their related classes in JDK. The sum of these two is the total time for Phase 1(a). The analysis time for the related classes in JDK is much longer than for those of the sample programs. For example, `TextEditor` itself requires only 10 milli seconds, and its related classes

[7]The UI subsystem uses Gtk−− tool kit [17].

Table 3: experimental results (computation time [ms])

Programs	Sample Program	Related Classes in JDK
TextEditor	10	14,224
JLex	892	3,863
java_cup	844	3,813
JFlex	16,140	14,339
WeirdX	2,835	14,666
ANTLR	6,154	7,856
Ant	1,845	4,005
DynamicJava	12,255	15,646

(a) Phase 1(a)

Programs	Sample Program	Related Classes in JDK
TextEditor	13	768
JLex	10	100
java_cup	10	99
JFlex	10	759
WeirdX	10	823
ANTLR	304	99
Ant	22	104
DynamicJava	1,892	843

(b) Phase 1(b)

Programs	Average
TextEditor	0.01
JLex	0.76
java_cup	0.37
JFlex	0.41
WeirdX	0.62
ANTLR	0.69
Ant	0.78
DynamicJava	0.07

(c) Phase 2

— Pentium4-2GHz-2GB(FreeBSD 4.6-STABLE)

Table 4: experimental results (Average number of detected aliases [expressions (VARIABLES)])

Programs (target class)	Instance-based	Class-based
TextEditor (`TextEditor`)	5.09(1)	5.09(1)
JLex (`JLex.CLexGen`)	69.17(2.02)	231.5(5.43)
java_cup (`java_cup.parser`)	101.6(1.48)	104.6(2.17)
JFlex (`JFlex.LexParse`)	124.2(1.50)	127.5(2.36)
WeirdX (`com.jcraft.weirdx.Client`)	68.33(2.97)	84.35(3.41)
ANTLR (`antlr.Tool`)	6.62(1.49)	11.37(2.30)
Ant (`org.apache.tools.ant.Main`)	20.94(3.25)	36.62(6.26)
DynamicJava (`koala.dynamicjava.interpreter.TypeChecker`)	9.16(1.89)	17.19(2.40)

require 14,224 milli seconds. When we modify `TextEditor`, we do not need to re-analyze its related classes, but only the `TextEditor`.

6.2.2 Computation Time of Phase 1(b)

TABLE 3(b) shows MFG construction time for sample programs and their related classes. Since MFG construction time does not depend on the program's size, but on the number of intra-class method calls, the MFG construction time of the sample program is not always longer than that of the related classes in JDK. For example, `DynamicJava` itself requires 1,892 milliseconds, but its related classes require only 843 milli seconds. However, the overall MFG construction time, the sum of these, is much smaller than the AFG construction time.

6.2.3 Computation Time of Phase 2

TABLE 3(c) shows the average AFG traversal time. According to TABLE 3(c), it is clear that Phase 2 takes much less computation time than Phase 1. In the case of `TextEditor`, 0.01 milli second is much smaller than the AFG construction time (14,234 milli seconds = 10 milli seconds + 14,224 milli seconds) for `TextEditor` and its related classes in JDK.

Our on-demand approach might be unsuitable as a back-end for data-flow analysis and compiler optimization, which needs whole alias analysis results. However, when we do not need to compute the aliases for all expressions, or when we implement an interactive programming support tool with alias analysis features, our method is a practical choice.

6.2.4 Average Number of Detected Aliases

The proposed method uses the *instance-based* approach that can distinguish inter-class alias relations on objects instantiated from the same class. On the other hand, if we use the *class-based* approach that shares inter-class alias relations with other objects instantiated from the same class, analysis precision will decrease.

On each testsuite, we select one main class (this does not mean the class which holds `public static int main()`, but the class which plays an dominant role in the testsuite), and we compute the aliases for each AFG normal node in that class.

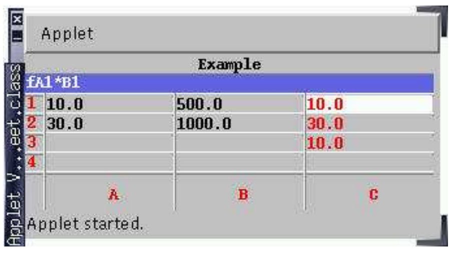
TABLE 4 shows the comparison results between those two approaches with regard to the average number of detected aliases for various alias criteria in the main classes. For example, the instance-based approach generates more accurate results than the class-based approach (9.16 nodes v.s. 17.19 nodes) in `DynamicJava`. The average size of aliases is about 30 – 97% of the class-based approach; therefore, we think that our approach is of practical value.

In some cases, the average number of detected aliases is not small. This is because we repeatedly count each expression, even if they have the same signature. If a specific variable is repeatedly used in a class, the number of the aliases becomes large. For example, in the case of `JLex`, the average number of the aliases is 69.17, for instance variable `JLex.CLexGen.m_outstream`, where `m_outstream` is referred to more than 400 times in the `JLex.CLexGen` class. However, the average number of unique variables in those aliases in `JLex.CLexGen` is only 1.50. This suggests that the users can easily focus their attention on only those few variables.

6.2.5 Application of JAAT

We focus on program maintenance activities as an application of JAAT. In order to examine JAAT's effectiveness, we have applied JAAT to the following program debugging case.

> `SpreadSheet.java` (1000 lines) is a small spreadsheet Java applet contained in JDK. We assume that an error occurred on the execution of `SpreadSheet.java` (Fig.15(a)). Since cell `C1` was defined `A1*B1`, `C1` should be 5000; however, `C1`'s value was incorrectly `10`.



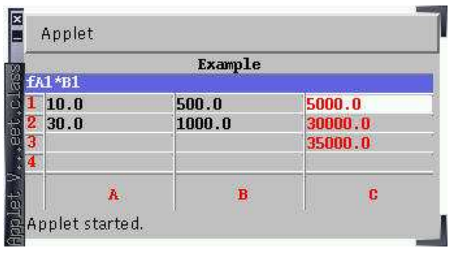
Example

fA1*B1

	A	B	C
1	10.0	500.0	10.0
2	30.0	1000.0	30.0
3			10.0
4			

Applet started.

(a) Incorrect output



Example

fA1*B1

	A	B	C
1	10.0	500.0	5000.0
2	30.0	1000.0	30000.0
3			35000.0
4			

Applet started.

(b) Correct output

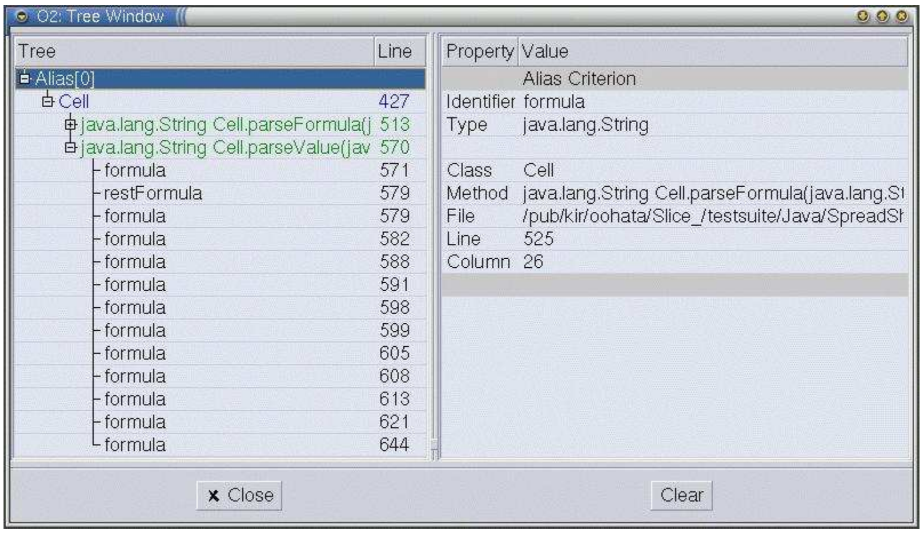
Figure 15: Example case: Output

In order to find the fault position, we tried to compute aliases for a `String`-type actual parameter `formula` in a method `parseFormula()`, which is a parser for the input expressions.

From the *alias tree window* [8] in Fig.16(a), we could consider that the resulting aliases are in `parseFormula()` and `parseValue()` only. We examined each expression on the alias tree using the alias tree window and the *text window* [9] as shown in Fig.16(b). After checking all aliases in `parseFormula()`, we noticed that the return expression in `parseValue()` is variable `formula`.

[8]The alias tree window shows an aliases tree, in which each node denotes class, method, or an expression which contains aliases.

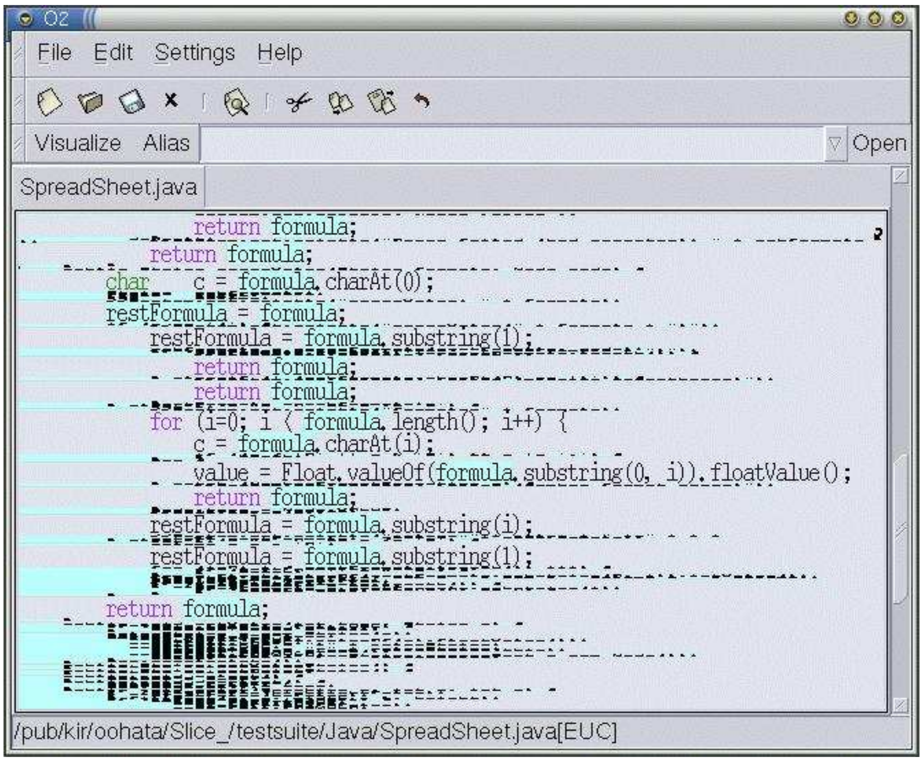
[9]The text window shows the resulting aliases with colored backgrounds. Statements without any aliases can be compressed on the screen with smaller fonts according to the user's request.



Tree	Line
Alias[0]	
Cell	427
java.lang.String Cell.parseFormula(j	513
java.lang.String Cell.parseValue(jav	570
formula	571
restFormula	579
formula	579
formula	582
formula	588
formula	591
formula	598
formula	599
formula	605
formula	608
formula	613
formula	621
formula	644

Property	Value
	Alias Criterion
Identifier	formula
Type	java.lang.String
Class	Cell
Method	java.lang.String Cell.parseFormula(java.lang.St
File	/pub/kir/oohata/Slice_/testsuite/Java/SpreadSh
Line	525
Column	26

(a) Alias tree window



```
                return formula;
            return formula;
        char    c = formula.charAt(0);
        restFormula = formula;
            restFormula = formula.substring(1);
                return formula;
                return formula;
            for (i=0; i < formula.length(); i++) {
                c = formula.charAt(i);
                value = Float.valueOf(formula.substring(0, i)).floatValue();
                return formula;
            restFormula = formula.substring(i);
            restFormula = formula.substring(1);
        return formula;
```

/pub/kir/oohata/Slice_/testsuite/Java/SpreadSheet.java[EUC]

(b) Text window

Figure 16: Example case: results

By examining several statements near the last return expression, we noticed that the return variable should be the variable `restFormula`, instead of `formula`. After fixing it, a new `SpreadSheet.java` was executed correctly (Fig.15(b)).

In this example, there are only 20 aliases, so that the user's attention can be focused on only 20 lines out of the original source program with about 1000 lines. The alias tree windows can help the user to grasp the overall resulting aliases, and the text windows can help the user to get detailed information about each alias. Using these windows, the user can efficiently perform program maintenance activities.

7 Discussion

Our proposed method for alias computation, which consists of two analysis phases, has produced effective results. In this section, we compare our method and related works, and also show an extension of our method to programs with pointer variables in ordinary languages.

7.1 Alias Analysis for Java

Analysis goal: Most prior studies on alias analysis for JAVA programs are for compiler optimization, such as *synchronization removal* and *escape analysis* [6, 7, 13, 32, 41]. For such purposes, all alias relations in the program need to be extracted by the analysis. Since the optimized program should compute the same execution results as the original program, analysis results must satisfy conservative approximation.

Nonetheless, we believe that alias analysis is useful for program maintenance activities as a software engineering tool. For such activities, not all the alias relations are needed at one time; only user-requested relations on the specific scope or object are to be extracted quickly. Also, good GUI that intuitively presents the analysis results to the user is very important.

Threads and exceptions: Since FS analysis considers the execution order of statements, its analysis precision depends on the precision of control-flow information. Thus, if we can collect more precise control-flow information, FS analysis results should become more precise.

Currently, since JAAT's control-flow representation does not consider possible control-flows caused by exceptions and threads, its resulting aliases contain surplus alias relations for exceptions and miss aliases caused by shared variables in threads. For exceptions, we applied a conservative approach that assumes all possible exceptions that might occur. However, since many researchers have already proposed control-flow analysis methods for threads and exceptions, we will adopt them to construct the improved control-flow representation [33, 36, 51].

Native methods: In JAVA, OS-dependent functions are implemented as *native methods*, such as the core parts of thread, I/O, reflection (dynamic class loading) and so on. Since JAAT does not concern native methods, its analysis results would be insufficient in a sense. However, since the user's focus is generally given to the user-developed parts of the program, current JAAT is enough to support program maintenance activities.

In order to consider native methods, we need the following approaches:

- Simulate the behavior of native methods.
- Implement a framework by which the user can provide helpful information to tools.

The former is useful in the case that tools can automatically infer the type of the related objects (for thread, I/O), the latter is useful in the case that tools need the user's help for type inference (for reflection).

7.2 Two-phase Approach

Several prior studies also propose two-phase approaches such as intra-procedural analysis in advance [11, 12, 18, 28]. Ramkrishna et al. have used a FS approach [11, 18]. Cheng et al. have used a FI approach [12, 28].

In the FS approach, each element $\mathcal{R}$ (alias relation) in an RAset holds conditions if a specific alias relation $\mathcal{R}_0$ really exists (if **true**, $\mathcal{R}$ exists) [11, 18]. These conditions are used for indirect alias relations; however, all combinations of accessible variables should be taken into account as candidates for $\mathcal{R}_0$. In our method, since each AFG edge represents a direct alias relation, we can easily extract each indirect alias relation as an AFG path. These conditional-based algorithms are suitable as back-end for data-flow analysis (e.g., program slicing), which requires whole alias relations in the target programs.

Since we focus mainly on program maintenance activities using the alias information itself, a simpler representation is useful. In such cases, the maintainers prefer the local alias information on which they focus. Also, we believe that they would request quick and simple answers even if the resulting aliases might be insufficient.

On the other hand, since our AFG traversal algorithm is designed to compute the aliases for the single alias criterion specified by the user, it is unsuitable for data-flow analysis (however, can define a new AFG traversal algorithm for computing whole alias relations in target programs).

Also, Cheng et al. target only ordinary procedural languages, and Ramkrishna et al. do not discuss on-demand analysis.

7.3 Instance-based Analysis

The instance-based approach was proposed in *object slicing*, which is a method for slicing OO programs proposed by Liang et al. [27]. They extend the *system dependence graph (SDG)* and define the traversal algorithm to compute slices with regard to a specific object; however, they assume that the alias analysis have been already performed by another method. To get a practical alias analysis tool, combining an alias analysis method and the instance separation method into a single method is very important, as we have proposed here.

7.4 On-demand Alias Extraction

We applied an on-demand approach to alias analysis. Although alias information has been used for compiler optimization, data-flow analysis and so on, alias information is itself useful for program maintenance activities such as program debugging and understanding of JAVA programs. JAVA programs generally have many aliases caused by reference-type expressions, and some are not easily identified by the developer. Since all alias information in the target program is not necessary on program maintenance activities, we believe on-demand (or query-based) analysis will be a cost-effective approach.

Although Heintze et al. have proposed a demand-driven pointer analysis, their approach is FI for procedural language C, and their goal is to compute the full point-to graph [19].

7.5 Selection of the Analysis Algorithm

Many prototype tools have been implemented for validation of their approaches [20, 22, 49]. However, there is little discussion on the extensibility of the algorithm. Since we have employed a phased approach for the implementation of JAAT, we can explore different algorithms by replacing each phase or sub-phase.

In Phase 1(a), we have used the FS approach for the analysis of a single method. Although FS analysis requires much larger computation time than the FI analysis, the intra-class analysis is generally a less costly process than the inter-class analysis. Thus, applying FS policy to the intra-method alias analysis is a practical choice.

In Phase 2, we have chosen an instance-based approach for generic inter-class analysis. This approach is efficient as well as effective in reducing the resulting aliases as shown in Section 4. Therefore, there is no reason to choose a class-based approach.

Here, we discuss other issues not yet discussed.

Object context: For computation of the object context at Phase 2, we have used the flow-insensitive object context (FIOC) approach rather than the flow-sensitive one (FSOC). We do not know the practical effectiveness of FSOC, but we would be able to extend the system to FSOC by introducing a notion of flow direction in AFG.

Undirected AFG and directed AFG: Our AFG construction algorithm is based on the FS alias algorithm [14, 26, 43]. However, since the current AFG is an undirected graph, the difference between the analysis computed by current AFG and traditional FS analysis exists in the sense that the current AFG does not hold an alias flow.

For example, in Fig.17(a), in traditional FS analysis, $\mathcal{A}(<s_1$, `a`$>)$ is $\{(s_1$, `a`$)$, $(s_1$, `new Integer(1)`$)\}$, and $\mathcal{A}(<s_2$, `b`$>)$ is $\{(s_2$, `b`$)$, $(s_2$, `a`$)$, $(s_1$, `a`$)$, $(s_1$, `new Integer(1)`$)\}$. However, in the current AFG, both $\mathcal{A}(<s_1$, `a`$>)$ and $\mathcal{A}(<s_2$, `b`$>)$ are $\{(s_2$, `b`$)$, $(s_2$, `a`$)$, $(s_1$, `a`$)$, $(s_1$, `new Integer(1)`$)\}$. Also, in Fig.17(b), due to two direct alias relations $\{(s_1$, `a`$)$, $(s_2$, `a`$)\}$, and $\{(s_1$, `a`$)$, $(s_3$, `a`$)\}$, an indirect alias relation $\{(s_2$, `b`$)$, $(s_3$, `c`$)\}$ is unwillingly generated.

```
1:   a = new Integer(1);
2:   b = a;
```

(a)

```
1:   a = new Integer(1);
2:   if(...) { b = a; }
3:   else { c = a; }
```

(b)

Figure 17: FS alias and undirected AFG

By introducing a notion of flow direction in AFG, the above problems are solved; however, we still need to consider the trade-off between the analysis precision and the analysis cost.

May-alias and must-alias: Distinguishing may-alias relations and must-alias relations is effective for more precise analysis. Applying this approach to AFG construction and traversal algorithms strengthens the system performance, but it increases the complexity of the system.

Heap-allocated storages: We can consider a special treatment for the heap-allocated storages, such as array variables and recursive structures. For array variables, the current AFG does not deal with each element separately as `A[0]`, `A[1]`, ... We can try to distinguish each element of array variables; however, by static analysis, the result would not be promising.

Recursive data structures: For recursive data structures such as *LinkedList* class with the attributes *data* and *next*, we can separately identify *L.data*, *L.next.data*, ..., *L.next.....data*, etc. (*L* is an instance of *LinkedList*) by our instance-based approach. Each attribute, however, cannot be distinguished if a class-based approach were to be employed. Landi et. al. have proposed *k-limiting* approach [25]; *k* is a *recursive-sensitivity parameter* that differentiates the first k objects in *LinkedList*. Our implemented instance-based approach corresponds to $k = 1$, and the class-based approach corresponds to $k = 0$.

7.6 Alias Analysis for Pointer Variables

In Section 4, we have described an alias analysis method for reference-type expressions in Java. This method has also been extended to the alias analysis of ordinary procedural programs with pointer variables such as C or C++. In this case, we need special consideration of indirect reference by pointers (e.g., multi-level pointers like `**p`) because a callee can modify its caller's alias relations using n-order (n $\geq$ 2) pointer variables even if parameter passing uses a passed-by-value mechanism. The detail is shown in our technical report [30].

8 Conclusions

We have proposed an alias analysis algorithm for Java programs, which is a scalable and on-demand algorithm with high precision and extensibility. Also, we have implemented this algorithm in the tool JAAT, and evaluated its effectiveness.

We are planning to implement the flow sensitive object context approach (FSOC), and compare this approach with the existing flow insensitive object context (FIOC) approach. In addition, we plan to extend our AFG construction and AFG traversal algorithms for full exception and thread compliance. Also, JAAT will be applied to existing software development activities, and the effectiveness of JAAT will be evaluated. The definition and implementation of an alias analysis algorithm for all alias relations simultaneously is also part of our future research.

We have shown that we can reduce analysis cost by our modularized approach where we re-analyze only modified modules, without re-analysis of other stable modules including libraries. In the future, we will include a feature to automatically identify modified modules and stable modules.

References

[1] G. Agrawal and L. Guo, *Evaluating explicitly context-sensitive program slicing*, Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, Pages 6–12, 2001, Snowbird, Utah, USA.

[2] A. V. Aho, S. Sethi and J. D. Ullman, *Compilers : Principles, Techniques, and Tools*, Addison-Weseley, 1986.

[3] L. O. Andersen, *Program Analysis and Specialization for the C Programming Language*, PhD thesis, 1994, DIKU, University of Copenhagen.

[4] *ANTLR Website*, `http://www.ANTLR.org/`.

[5] D. C. Atkison and W. G. Griswold, *The Design of Whole-Program Analysis Tools*, Proceedings of the 18th International Conference on Software Engineering, Pages 16–27, 1996, Berlin, Germany.

[6] B. Blanchet, *Escape Analysis for Object-Oriented Languages: Application to Java*, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, Pages 20–34, 1999, Denver, Colorado, USA.

[7] J. Bogda, U. Holzle, *Removing Unnecessary Synchronization in Java*, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, Pages 35–46, 1999, Denver, Colorado, USA.

[8] G. Booch, *Object-Oriented Design with Application*, The Benjamin/Cummings Publishing Company, Inc, 1991.

[9] M. Burke, P. Carini, J. D. Choi and M. Hind, *Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers*, Proceedings from the 7th International Workshop on Languages and Compilers for Parallel Computing, 1994, Ithaca, New York, USA.

[10] D. R. Chase, M. N. Wegman and F. K. Zadeck, *Analysis of Pointers and Structures*, Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, Pages 296–310, 1990, White Plains, New York, USA.

[11] R. K. Chatterjee, B. G. Ryder and W. Landi, *Relevant Context Inference*, Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages, Pages 133–146, 1999, San Antonio, Texas, USA.

[12] B. C. Chenga and W. W. Hwu, *Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation*, Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, Pages 57–69, 2000, Vancouver, Britith Columbia, Canada.

[13] J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar and S. P. Midkiff, *Escape Analysis for Java*, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, Pages 1-19, 1999, Denver, Colorado, USA.

[14] M. Enami, R. Ghiya and L. J. Hendren, *Context-sensitive interprocedural points-to analysis in the presence of function pointers*, Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, Pages 242–256, 1994, Orlando, Florida, USA.

[15] *Extensible Markup Language(XML)*, `http://www.w3c.org/XML/`.

[16] J. Gosling, B. Joy and G. Steele, *The* JAVA^{TM} *Language Specification*, Addison-Weseley, 1996.

[17] *Gtk−−*, `http://gtkmm.sourceforge.net/`.

[18] M. J. Harrold and G. Rothermel, *Separate Computation of Alias Information for Reuse*, IEEE Transactions on Software Engineering, Special section of best papers of the 1996 International Symposium on Software Testing and Analysis, Vol. 22, No. 7, Pages 442–460, 1996.

[19] N. Heintze and O. Tardieu, *Demand-Driven Pointer Analysis*, Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation, Pages 24–34, 2001, Snowbird, Utah, USA.

[20] M. Hind and A. Pioli, *Assessing the Effects of Flow-Sensitivity on Pointer Alias Analysis*, Proceedings of the 5th International Static Analysis Symposium, Pages 57–81, 1998, Pisa, Italy.

[21] M. Hind, M. Burke, P. Carini and J. D. Choi, *Interprocedural Pointer Alias Analysis*, ACM Transactions on Programming Languages and Systems, Vol. 21, No. 4, Pages 848–894, 1999.

[22] M. Hind and A. Pioli, *Which Pointer Analysis Should I Use?*, Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Pages 22–25, 2000, Portland, Oregon.

[23] M. Hind, *Pointer Analysis: Haven't We Solved This Problem Yet?*, Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Pages 54–61, 2001, Snowbird, Utah.

[24] T. Kamiya, F. Ohata, K. Kondo, S. Kusumoto and K. Inoue, *Maintenance support tools for Java programs: CCFinder and JAAT*, Proceedings of the 23th International Conference on Software Engineering, Pages 837–838, Formal Research Demonstrations, 2001, Toronto, Canada.

[25] W. Landi and B. G. Ryder, *A Safe Approximate Algorithm for Interprocedural Pointer Aliasing*, Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation, Pages 235–248, 1992, San Francisco, California, USA.

[26] W. Landi, B. G. Ryder and S. Zhang, *Interprocedural Modification Side Effect Analysis With Pointer Aliasing*, Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, Pages 56–67, 1993, Albuquerque, New Mexico, USA.

[27] D. Liang and M. J. Harrold, *Slicing Objects Using System Dependence Graphs*, Proceedings of the International Conference on Software Maintenance, Pages 358–367, 1998, Washington, D.C., USA.

[28] D. Liang and M. J. Harrold, *Efficient Points-to Analysis for Whole-Program Analysis*, Proceedings of the 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Pages 199–215, 1999, Toulouse, France, September 1999.

[29] F. Ohata and K. Inoue, *Alias analysis for object-oriented programs*, Technical Report on IPSJ-SIGSE-126, Vol. 2000, No. 25, 2000-SE-126, Pages 57–64 (in Japanese).

[30] F. Ohata and K. Inoue, *JAAT: A Practical Alias Analysis Tool for Java Programs*, Technical Report of Osaka University, Department of Information and Computer Sciences, IIP Lab, IIP-12-27-01, Dec 27, 2001.

[31] H. D. Pande and B. G. Ryder, *Data-Flow-Based Virtual Function Resolution*, Proceedings of the Third International Static Analysis Symposium, Pages 238–254, 1996, Aachen, Germany.

[32] A. Rountev, A. Milanova and B. G. Ryder, *Points-To Analysis for Java using Annotated Constraints*, Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Pages 43–55, 2001, Tampa, Florida, USA.

[33] R. Rugina and M. C. Rinard, *Pointer Analysis for Multithreaded Programs*, Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation, Pages 77–90, 1999, Atlanta, Georgia, USA.

[34] B. G. Ryder, W. Landi, P. Stocks, S. Zhang and R. Altucher, *A schema for interprocedural modification side-effect analysis with pointer aliasing*, ACM Transactions on Programming Languages and Systems, Vol. 23, No. 2, Pages 105–186, 2001.

[35] M. Shapiro and S. Horwitz, *Fast and accurate flow-insensitive point-to analysis*, Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Pages 1–14, 1997, Paris, France.

[36] S. Sinha and M. J. Harrold, *Analysis of Programs With Exception-Handling Constructs*, Proceedings of the International Conference on Software Maintenance, Pages 358–367, 1998, Washington, D.C., USA.

[37] B. Steensgaard, *Points-to analysis in almost linear time*, Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Pages 32–41, 1996, St. Petersburg Beach, Florida, USA.

[38] B. Stroustrup, *The C++ Programming Language(Third edition)*, Addison-Wesley, 1997.

[39] F. Tip and J. Palsberg, *Scalable propagation-based call graph construction algorithms*, Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, Pages. 281–293, 2000, Minneapolis, Minnesota, USA.

[40] P. Tonella, G. Antoniol, R. Fiutem and E. Merlo, *Flow insensitive C++ pointers and polymorphism analysis and its application to slicing*, Proceedings of the 19th International Conference on Software Engineering, Pages 433–443, 1997, Boston, Massachusetts, USA.

[41] F. Vivien, M. C. Rinard, *Incrementalized Pointer and Escape Analysis*, Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation, Pages 35–46, 2001, Snowbird, Utah, USA.

[42] M. Weiser, *Program slicing*, Proceedings of the 5th International Conference on Software Engineering, Pages 439–449, 1981, San Diego, California, USA.

[43] R. P. Wilson and M. S. Lam, *Efficient context-sensitive pointer analysis for C programs*, Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, Pages 1–12, 1995, La Jolla, California, USA.

[44] *The Wisconsin Program-Slicing Tool 1.0, Reference Manual*, Computer Sciences Department, University of Wisconsin-Madison, 1997.

[45] *The XML C library for Gnome*, `http://xmlsoft.org/`.

[46] *Xalan-C++*, `http://xml.apache.org/xalan-c/index.html`.

[47] *XSL Transformations (XSLT) Specification*, `http://www.w3.org/TR/1999/WD-xslt-19990421`.

[48] S. H. Yong, S. Horwitz and T. W. Reps, *Pointer Analysis for Programs with Structures and Casting*, Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation, Pages 91–103, 1999, Atlanta, Georgia, USA.

[49] S. Zhang, B. G. Ryder and W. A. Landi, *Experiments with Combined Analysis for Pointer Aliasing*, Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Pages 11–18, 1998, Montreal, Canada.

[50] S. Zhang, B. G. Ryder and W. A. Landi, *Program Decomposition for Pointer Aliasing: A Step Toward Practical Analyses*, Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, Pages 81–92, 1996, San Francisco, California, USA.

[51] J. Zhao, *Slicing Concurrent Java Programs*, Proceedings of the 7th IEEE International Workshop on Program Comprehension, Pages 126–133, 1999, Pittsburgh, Pennsylvania, USA.