# Constructing Accurate Application Call Graphs For Java To Model Library Callbacks

Weilei Zhang, Barbara Ryder
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
{weileiz,ryder}@cs.rutgers.edu

## Abstract

*Call graphs are widely used to represent calling relationships among methods. However, there is not much interest in calling relationships among library methods in many software engineering applications such as program understanding and testing, especially when the library is very big and the calling relationships are not trivial. This paper explores approaches to generate more accurate application call graphs for Java. A new data reachability algorithm is proposed and fine-tuned to resolve library callbacks accurately. Compared to a simple algorithm that generates an application call graph by traversing the whole-program call graph, the fine-tuned data reachability algorithm results in fewer spurious callback edges. In experiments with the spec jvm98 benchmarks, the new algorithm shows a significant reduction in the number of spurious callback edges over the simple algorithm: on average, the number of callback edges is reduced by 74.97%, amounting to overall 64.43% edge reduction for the generated application call graphs.*

## 1 Introduction

A *call graph* is the representation of calling relationships among methods: a directed edge from method *a* to *b* denotes that *a* may call *b* directly. Call graphs are widely used as a program representation in software engineering and optimizing compilation. Construction of call graphs is usually straightforward in classical procedural languages; for example, in *C*, barring the use of function pointers, a call site has exactly one possible callee. In object-oriented languages, a call site may invoke several callees due to dynamic dispatch. The corresponding call graph construction ([4]) uses some form of *reference analysis*. Reference analysis calculates type information about the objects to which reference variables can point. There is a wide variety of reference analyses which differ in terms of cost and precision. An in-depth discussion can be found in [12, 5].

Precise reference analysis requires a whole-program analysis. The constructed call graph includes both *application* and *library* methods as its nodes. However for many software engineering applications such as program understanding and testing, there is not much interest in the calling relationships among library methods. In these contexts, an accurate *application call graph* is more useful than a whole-program call graph. Also, a static analysis requiring a call graph can run more efficiently and produce more accurate results for application program if an accurate application call graph can be substituted for a whole-program call graph.

**Application Call Graph.**

An application call graph represents calling relationships among application methods. There are two kinds of edges: $direct$ and $callback$. For application methods $a$ and $b$, a $direct$ edge from $a$ to $b$ means that there is a call site in *a* that resolves to a call of *b*. A $callback$ edge from $a$ to $b$ means that *a* may call back *b* through the library; that is to say, *a* may call a library method that may eventually call *b*, and there exists a call path from *a* to *b*, $a \rightarrow m_1 \rightarrow m_2 \rightarrow ... \rightarrow m_n \rightarrow b$ on which all the intermediate methods ($m_i$) are library methods. Call edges in an application call graph may have labels to denote call site information. For example, a $callback$ edge from $a$ to $b$ with label $s$ means that at statement $s$, method $a$ makes a library call, from which it may eventually call back $b$; we say that "call site $s$ calls back $b$" for brevity.

The contributions of this work are:

- Design of new approaches to construct an accurate *application call graph* for Java. A new variant of the data reachability algorithm ([3]) is proposed and fine tuned to resolve library callbacks accurately.
- Implementation of the proposed algorithm and experiments with it.

- Description of the potential usages of application call graphs in white-box testing.

**Outline.**

The rest of the paper is organized as follows: Section 2 discusses a simple algorithm to generate application call graphs by traversing whole-program call graphs. Section 3 presents an algorithm to resolve library callbacks accurately. Section 4 describes the empirical study. Section 5 discusses the potential usages of application call graphs generated by the given algorithm. Section 6 discusses related work. Section 7 gives conclusions and directions for future work.

## 2 A Simple Algorithm And Its Imprecision

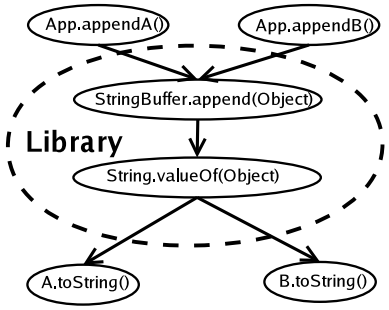
After a whole-program call graph is generated by using some form of reference analysis, an application call graph can be generated by traversing the whole-program call graph. A $direct$ call edge is generated if there is a call edge between two application methods in the whole-program call graph. A $callback$ edge is generated between a pair of application methods if there is a directed path between them in the whole-program call graph on which all intermediate nodes are library methods.

The application call graph generated by the above simple algorithm represents the calling relationships among application methods that can be captured by the whole-program call graph. There is a one-to-one mapping between $direct$ edges and call edges among application methods in the whole-program call graph, so the precision for $direct$ edges corresponds directly to the precision for the whole-program call graph([4]). $Callback$ edges are generated by collapsing through-library call paths that connect a pair of application methods in the whole-program call graph. Many such through-library call paths cannot happen at runtime; consequently, the corresponding $callback$ edges generated by the simple algorithm are spurious.

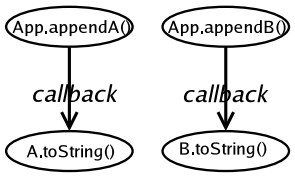
Figure 1 shows an example to illustrate the simple algorithm and its imprecision. Figure 1-(a) is a piece of Java code, and 1-(b) shows part of the corresponding whole-program call graph. `App.appendA()` and `App.appendB()` are two application methods both calling the library method `StringBuffer.append(Object)` at call sites $(5)$ and $(9)$, respectively. Classes `StringBuffer` and `String` both come from the `java.lang` library package. `StringBuffer.append(Object)` calls `String.valueOf(Object)`, which in turn calls a `toString()` method. If `r` is the actual parameter passed to `StringBuffer.append(Object)`, then `String.valueOf(Object)` will call `Object.toString()` on the object pointed to by `r`. In this example at call site $(5)$, `r` points to the $A$ object created at $(4)$, and class $A$ overrides the `toString()` method,

```
(1)  class App{
(2)    StringBuffer local;
(3)    StringBuffer appendA(){
         A a=new A();
(5)      return(local.append(a));
(6)    }
(7)     StringBuffer appendB(){
(8)       B b=new B();
(9)       return(local.append(b));
(10)    }
(11)  }
(12)  class A{
(13)    String toString(){...}
(14)  }
(15)  class B{
(16)    String toString(){...}
(17)  }
```

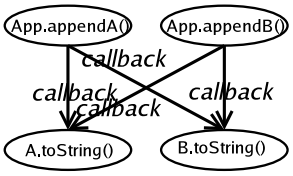
(a) Code




(b) Whole-program Call Graph




(c) Precise Application Call Graph




(d) Application Call Graph Generated by the Simple Algorithm

**Figure 1. A Example to Illustrate the Simple Algorithm and its Imprecision**

so `App.appendA()` will call back `A.toString()` at runtime. Similarly, `App.appendB()` will call back `B.toString()`. Consequently, an accurate application call graph should look like Figure 1-(c), in which there are two *callback* edges. But in Figure 1-(b), there is a directed path from `App.appendB()` to `A.toString()`, and all intermediate nodes, `StringBuffer.append(Object)` and `String.valueOf(Object)`, are library methods. According to the simple algorithm, a spurious $callback$ edge will be generated from `App.appendB()` to `A.toString()`; similarly, another spurious $callback$ edge will be generated from `App.appendA()` to `B.toString()`, as shown in Figure 1-(d).

The above problem exists because the whole-program call graph lacks calling context information. The shared segments (in this case the path from `StringBuffer.append(Object)` to `String.valueOf(Object)`) result in infeasible call paths connecting different start and end points. This problem cannot be completely solved unless enough context information is added to the call graph construction algorithm. For example, the specific problem in Figure 1 can be solved by 2-CFA [14, 15], but 2-CFA is very expensive for whole-program analysis. What's more, in many cases the length of the shared segment is much longer than 2. The overarching problem will require $n$-CFA with a very large $n$ or the use of a call tree [13] instead of a call graph to represent calling relationships. Currently, both approaches are impractical because they are not scalable for real-world programs.

# 3 A Data Reachability Algorithm To Resolve Library Callbacks

We want to use a rather precise yet practical analysis to eliminate as many infeasible through-library call paths as possible, to reduce the number of spurious $callback$ edges in the generated application call graph. The data reachability algorithm ([3]) is used to solve this problem. In this section, we begin by introducing the data reachability algorithm. Then a new variant of data reachability, $V^a$-$DataReach$, is proposed and compared to the existing $V$-$DataReach$ algorithm. Finally the algorithm is fine-tuned specifically to resolve library callbacks more accurately, resulting in $V^a$-$DataReach^{ft}$.

## 3.1 Data Reachability Algorithm

The intuitive idea of the data reachability algorithm is to resolve control-flow reachability (i.e., find feasible call paths) via data reachability analysis. Call paths requiring receiver objects of a specific type can be shown to be infeasible, if those types of objects are not reachable through dereferences at the relevant call site. In Figure 1, the call path `App.appendA()` $\rightarrow$ `StringBuffer.append(Object)` $\rightarrow$ `String.valueOf(Object)` $\rightarrow$ `B.toString()` is feasible, only if *during the lifetime* of the library call `StringBuffer.append(Object)` at call site (5), the receiver object of the site calling `Object.toString()` inside the method `String.valueOf(Object)` can be of type $B$; if this cannot happen, then the above call path is infeasible.

Fu, *et. al* present three forms of data reachability algorithms in [3]: *DataReach*, *M-DataReach* and *V-DataReach*, listed in order of accuracy of their solutions. *DataReach* uses one set to record all possible reachable objects during the lifetime of a specific method call. *M-DataReach* uses a separate set for each method to record that method's possible reachable objects during the lifetime of a specific method call. *V-DataReach* uses a separate set for each reference variable and each object field to record its possible referenced objects during the lifetime of a specific method call.

In essence, the data reachability algorithm performs a separate reference analysis for each call site after a whole-program reference analysis. More specifically for *V-DataReach*, there are two kinds of points-to analyses in the algorithm: one is a whole-program analysis, and the other is a call-site specific analysis. During the call-site specific points-to analysis, an object is either $accessible$ or $local$. $Accessible$ means that **before** the end of the call, the object may be accessed from code executed outside the reachable methods of this method call (e.g., through another thread). Consequently, for an instance field read statement $l = r.f$[1] encountered during the call-site specific analysis, if $r$ points to an $accessible$ object $o$, it means that $o.f$ may have been changed elsewhere, so the global points-to result for $o.f$ is used in the call-site specific points-to analysis. In $V$-$DataReach$, in order to calculate the set of those $accessible$ objects, a global $escape$ analysis ([2]) is performed after the whole-program points-to analysis and before the call-site specific analysis. If an object may escape the method that creates it according to the escape analysis, it is considered $accessible$ in $V$-$DataReach$. In this paper, we propose a new variation of the data reachability algorithm: $V^a$-$DataReach$, that differs from $V$-$DataReach$ by calculating the set of *accessible* objects *on the fly* during the process of calculating the set of methods reachable from a call, using a call-site specific points-to analysis.

## 3.2 $V^a$-DataReach.

Similarly to $V$-$DataReach$, $V^a$-$DataReach$ needs an initial whole-program points-to analysis, whose result is denoted as $Pt$. For a given call site, the algorithm computes

[1]For brevity, we omit the cases for static fields and arrays in our discussion. The static fields can be considered to belong to a single (fake) object that is $accessible$. An array instance is regarded as one object and all accesses to elements of this array are modelled using a single field.

the set of $accessible$ objects ($Accessible$), the call-site specific points-to result ($U$) and the set of reachable methods ($R$). If needed, the reachable sub-call graph can be also computed.

Both of the points-to analysis results, $Pt$ and $U$, contain points-to information ($\mathcal{P}(O)$) for each reference variable ($Ref$) and object field ($O \times F$), where $O$ is the set of object creation sites and $F$ is the set of object fields. $U$ is a subset of $Pt$. During the call-site specific analysis to calculate $U$, the points-to information for the fields of the $accessible$ objects comes from $Pt$, while the points-to information for the local reference variables and the fields of the $local$ objects comes from $U$.

An object $o$ is $accessible$ if it satisfies one of the following:

- $o$ is referenced by an actual parameter passed to the call site.
- $o$ is referenced by a static field.
- $o$ is reachable from an $accessible$ object $a$ through field access (i.e., there exists a list of object fields $f_i$s such that $a.f_1.f_2.....f_n$ refers to $o$).

In $V^a$-$DataReach$, the set of $accessible$ objects is calculated on the fly during the call-site specific points-to analysis. For example, if an instance field read statement $l = r.f$ is encountered, and if $r$ points to an $accessible$ object $o$, both $U_l$ and $Accessible$ will be updated and $Pt(o.f)$ will be included in $Accessible$ (see constraint 4 below).

$V^a$-$DataReach$ is defined by the following constraints, using the constraint-based formalism from [18], analogous to the data reachability algorithm schema defined in [3]:

- **input:**
  $\begin{cases} Pt : Ref \rightarrow \mathcal{P}(O), O \times F \rightarrow \mathcal{P}(O) \\ \text{the original call site as the starting point.} \end{cases}$
- **output:**
  $\begin{cases} R \\ Accessible \\ U : Ref \rightarrow \mathcal{P}(O), O \times F \rightarrow \mathcal{P}(O) \end{cases}$
- **initialize:** for each target $M$ at original call and the corresponding actuals $a_i$ and formals $M.f_i$ :
  $\begin{cases} M \in R \wedge \\ Pt(a_i) \subseteq Accessible \wedge \\ Pt(a_i) \subseteq U_{M.f_i} \end{cases}$
  Initialize $U_{M.this}$ of targets $M$ accordingly
  Initialize all other $U_v$ and $U_{o.f}$ to $\emptyset$

1. For each method $M$ and for each object creation statement $s_i$: $l = new\ o_i$ in $M$:
   $(M \in R) \Rightarrow o_i \in U_l$
2. For each method $M$ and for each reference assignment statement $s_i$: $l = r$ in $M$:
   $(M \in R) \Rightarrow U_r \subseteq U_l$
3. For each method $M$, and for each instance field write statement $l.f = r$ in $M$ and each $o_i \in Pt(l)$:
   $(M \in R) \wedge (o_i \in U_l) \Rightarrow$
   $\begin{cases} o_i \notin Accessible \Rightarrow U_r \subseteq U_{o_i.f} \\ o_i \in Accessible \Rightarrow U_r \subseteq Accessible \end{cases}$
4. For each method $M$, and for each instance field read statement $l = r.f$ in $M$ and each $o_i \in Pt(r)$:
   $(M \in R) \wedge (o_i \in U_r) \Rightarrow$
   $\begin{cases} o_i \notin Accessible \Rightarrow U_{o_i.f} \subseteq U_l \\ o_i \in Accessible \Rightarrow \begin{cases} Pt(o_i.f) \subseteq U_l \wedge \\ Pt(o_i.f) \subseteq Accessible \end{cases} \end{cases}$
5. For each method $M$, for each virtual call site $l = e.m(e_1, \ldots, e_n)$ occurring in $M$, and for each $o \in Pt(e)$ where $StaticLookup(o, m) = M'$:
   $(M \in R) \wedge (o \in U_e) \Rightarrow$
   $\begin{cases} M' \in R \wedge \\ U_{e_i} \subseteq U_{M'.f_i} \text{ where } f_i \text{ are the formal parameters of } M' \wedge \\ U_{M'.ret\_var} \subseteq U_l \wedge \\ o \in U_{M'.this} \end{cases}$
6. For each method $M$ and for each static field write statement $C.f = l$ in $M$:
   $(M \in R) \Rightarrow U_l \subseteq Accessible$
7. For each method $M$ and for each static field read statement $l = C.f$ in $M$:
   $(M \in R) \Rightarrow$
   $\begin{cases} Pt(C.f) \subseteq U_l \\ Pt(C.f) \subseteq Accessible \end{cases}$
8. For each method $M$ and for each static call site $l = C.M'(e_1, \ldots, e_n)$ in $M$:
   $(M \in R) \Rightarrow$
   $\begin{cases} M' \in R \wedge \\ U_{e_i} \subseteq U_{p_i} \text{ where } p_i \text{ are the formal parameters of } M' \wedge \\ U_{M'.ret\_var} \subseteq U_l \end{cases}$

During initialization, $V^a$-$DataReach$ populates $U$ and $Accessible$ according to the whole-program points-to information for the corresponding actual parameters, and initializes $R$ to include the possible target methods of the original call site. Constraints 1 and 2 handle object creation and reference assignment statements and update $U$ accordingly. Constraint 3 handles the instance field write statement $l.f = r$: for an object $o_i$ pointed to by $l$, if $o_i$ is $local$, then $U_{o_i.f}$ is updated by $U_r$. $U_{o_i}$ need not be updated when $o_i$ is $accessible$ because the whole-program points-to information will be used for $o_i$; also, objects in $U_r$ will be marked as $accessible$ if $o_i$ is $accessible$. Constraint 4 handles the instance field read statement $l = r.f$: if $r$ refers to an $accessible$ object $o_i$, the result of the whole-program points-to analysis for $o_i.f$ will be used to update $U_l$ and $Accessible$; otherwise ($o_i$ is $local$), $Accessible$ remains unchanged, and the result of the call-site specific points-to analysis for $o_i.f$ will be used to update $U_l$. Constraint 5 specifies the addition of new methods to the set of reachable methods at virtual calls: a new method $M'$ is added

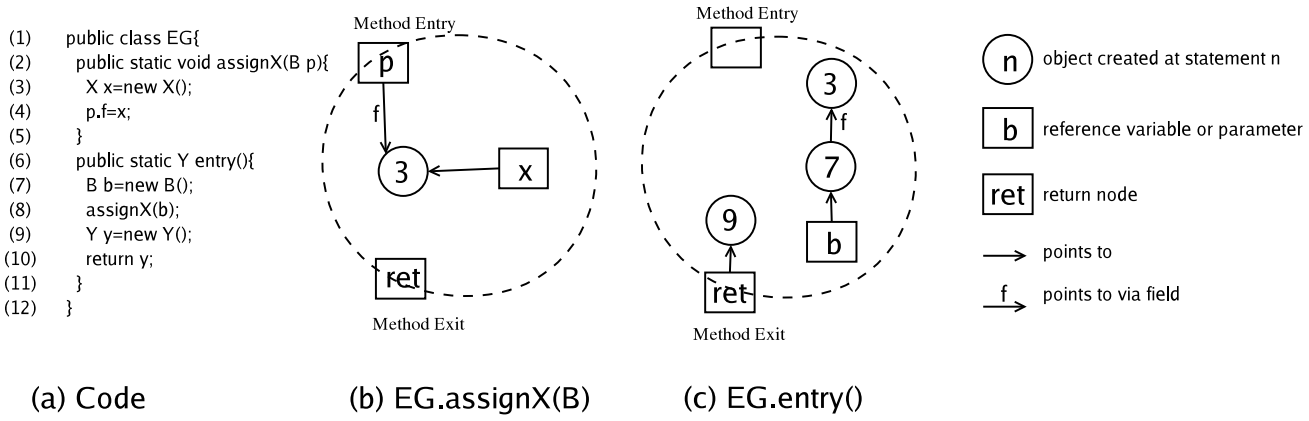



**Figure 2. An Example to Illustrate the Difference between V$^a$-DataReach vs. V-DataReach**

to $R$ only if the required object(s) to trigger the invocation of $M'$ are in the call-site specific points-to set of the receiver reference variable. $U$ is modified because of parameter assignments and the return value. The auxiliary function $StaticLookup$ returns the dynamic dispatch target of virtual call, given the receiver object and the compile-time target method. Constraints 6, 7 and 8 handle static field writes, static field reads and static call sites, respectively.

**Comparison: V$^a$-DataReach vs. V-DataReach.**

$V$-$DataReach$ and $V^a$-$DataReach$ calculate the set of $accessible$ objects differently. $V^a$-$DataReach$ calculates the set on the fly as shown in the constraints. In contrast, $V$-$DataReach$ requires the result of a separate escape analysis, and considers an object $accessible$ if the object may escape the method that creates it. Figure 2 illustrates the difference between both algorithms. Figure 2-(a) is a piece of Java code that contains two methods: `EG.entry()` and `EG.assignX(B)`. Figures 2-(b),(c) illustrate the points-to graph for the two methods, in which we use the statement sequence number to represent the object created at that creation statement. Assume that we apply the two data reachability algorithms to the same call site that calls `EG.entry()`. Because Object 3 is referenced via a field from the parameter of method `EG.assignX(B)`, which creates it, it escapes `EG.assignX(B)`, and thus is regarded as $accessible$ by $V$-$DataReach$. In contrast, it can be seen from Figure 2-(c) that Object 3 is not $accessible$ from the code executed beyond the method call of `EG.assignX(B)`, and thus is regarded as $local$ by $V^a$-$DataReach$. Another example is Object 9: it also escapes the method creating it via the return node and thus is regarded as $accessible$ by $V$-$DataReach$; but it is not $accessible$ from the code executed beyond the method call to `EG.assignX(B)` until **after** the call finishes and returns, so it will not be considered $accessible$ by $V^a$-$DataReach$.

In both $V^a$-$DataReach$ and $V$-$DataReach$ the points-to information for fields of $accessible$ objects comes from $Pt$, while the points-to information for fields of $local$ objects comes from $U$. $U$ is a subset of $Pt$, so the fewer the number of $accessible$ objects, the more accurate the data reachability algorithm result can be. In the example shown in Figure 2, two fewer objects, 3 and 9 are considered $accessible$ in $V^a$-$DataReach$ than in $V$-$DataReach$, so $V^a$-$DataReach$ can get more accurate results than $V$-$DataReach$. However, it is hard to draw a general conclusion from this simple example. As part of future work, we will examine the difference between the two notions of $accessible$ objects and its influence on the accuracy of the data reachability algorithms, through empirical studies.

**Using V$^a$-DataReach to Resolve Library Callbacks**

If in constraints 5 and 8 of $V^a$-$DataReach$, the reachable call edge $< M, cs, M' >$ is recorded for each reached method $M'$ at call site $cs$ in $M$, then a sub-call graph reachable from a specific call site can be generated. Given a library call, $libcall$, and the sub-call graph reachable from it generated by $V^a$-$DataReach$, $callback$ edges can be resolved in a similar way as in the simple algorithm: if there is a call path from $libcall$ to an application method $am$, and all the intermediate nodes on the path are library methods, then $libcall$ calls back $am$. The application call graph can be formed using these $callback$ edges found from each library call plus the $direct$ call edges found by the whole-program points-to analysis.

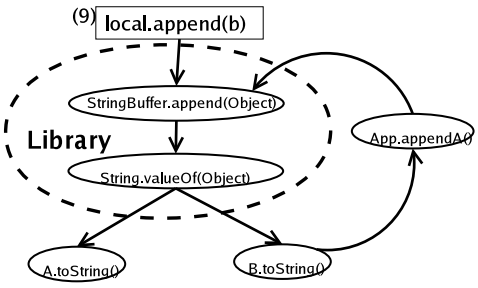
### 3.3 V$^a$-DataReach$^{ft}$: Fine-Tuned Algorithm To Resolve Library Callbacks

To calculate $callback$ edges for each library call from an application method, the data reachability algorithm needs some fine tuning to increase accuracy, as illustrated in Figure 3.

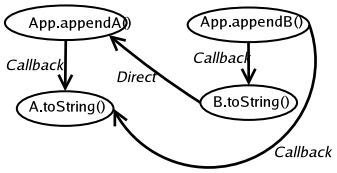
Figure 3-(a) is a slight modification of Figure 1-(a), where method `B.toString()` contains one more state-

```
(1)  class App{
(2)    StringBuffer local;
(3)    StringBuffer appendA(){
         A a=new A();
(5)      return(local.append(a));
(6)    }
(7)    StringBuffer appendB(){
(8)      B b=new B();
(9)      return(local.append(b));
(10)   }
(11)  }
(12)  class A{
(13)   String toString(){...}
(14)  }
(18)   class B{
         String toString(){
(20)       new App().appendA();
(21)       ......}
(22)   }
```

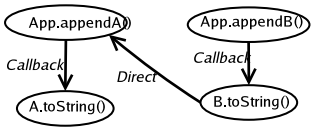
(a) Code For B.toString()




(b) Discoverd Sub Call Graph While Running General Data Reachability Algorithm On Call Site (9) In Figure 3-(a)




(c) Application Call Graph Generated by $^{a}V$ -DataReach




(d) Actual Application Call Graph

**Figure 3. An Example to Illustrate the Need for Fine-Tuned Algorithm**

ment in line 20. Originally in Figure 1, $V^a$-$DataReach$ determines that call site $(9)$ of method `App.appendB()` calls back `B.toString()` only. But the new codes in Figure 3-(a) introduce the following complication: at call site $(20)$, method `B.toString()` calls `App.appendA()`, which in turn calls back `A.toString()`. Figure 3-(b) shows the discovered sub-call graph by running $V^a$-$DataReach$ on call site $(9)$: both `A.toString()` and `B.toString()` show up, and it is hard to distinguish `A.toString()` from `B.toString()` while generating $callback$ edges for call site $(9)$ of method `App.appendB()`. Figure 3-(c) shows the application call graph generated by $V^a$-$DataReach$. Compared to the actual application call graph shown in Figure 3-(d), one spurious $callback$ edge from `App.appendB()` to `A.toString()` is generated.

In order to solve this problem, we propose $V^a$-$DataReach^{ft}$ based on $V^a$-$DataReach$. The intuition is that only library methods are included in $R$ during the call-site specific points-to analysis. The following is the substitute for constraint 5 in $V^a$-$DataReach$ to handle virtual call sites:

5'. For each method $M \in Lib$, for each virtual call site $l = e.m(e_1, \ldots, e_n)$ occurring in $M$, and for each $o \in Pt(e)$ where $StaticLookup(o, m) = M'$ and $f_i$ are the formal parameters of $M'$:

$$(M \in R) \wedge (o \in U_e) \Rightarrow \begin{cases} M' \in Lib \Rightarrow \begin{cases} M' \in R \wedge \\ U_{e_i} \subseteq U_{M'.f_i} \wedge \\ U_{M'.ret\_var} \subseteq U_l \wedge \\ o \in U_{M'.this} \end{cases} \\ M' \notin Lib \Rightarrow \begin{cases} M' \in Callback \wedge \\ U_{e_i} \subseteq Accessible \wedge \\ Pt(M'.ret\_var) \subseteq U_l \wedge \\ Pt(M'.ret\_var) \subseteq Accessible \end{cases} \end{cases}$$

The target method $M'$ of a virtual call site is added to the set $R$ only if $M'$ is a library method. If not, $M'$ is added to the $Callback$ set. Also, the objects referenced by the parameters passed to $M'$ or returned by $M'$ are $accessible$ from the code executed beyond this library entry before this library call finishes. There is also a similar substitute for constraint 8 in $V^a$-$DataReach$ to handle static call site. For brevity, it is not shown here.

## 4 Empirical Study

We have implemented two algorithms to generate the application call graph. The first one, denoted as $simple$, generates the application call graph by traversing the whole-program call graph generated by a *0-CFA* points-to analysis [11, 8]. The second one, denoted as $new$, starts from the results of the same *0-CFA* analysis, and resolves library callbacks according to $V^a$-$DataReach^{ft}$ presented in Section 3. This section describes our experiments with the two

algorithms. We aim to answer the following two questions in the experimental study:

- Accuracy: how many spurious $callback$ edges can be eliminated by $V^a$-$DataReach^{ft}$ from those generated by the simple algorithm?
- Practicality: the advantage for the simple algorithm is that it is cheap after a whole-program call graph is generated. Comparatively, is $V^a$-$DataReach^{ft}$ practical enough? Is it also scalable?

### 4.1 Experiment Setup.

We experimented on all eight benchmarks in the SPEC jvm98 suite ([17]). All experiments were run on a 1.8GHz AMD Athlon(tm) 64 Processor 3000+, 2GB-memory PC with Linux 2.6.12-gentoo-r10 and Sun JVM 1.4.1.07 (32-bit). The algorithm is implemented in the framework presented in [19], which utilizes a Java optimization framework, *Soot* ([7]) and a BDD-based constraint solver, *bdddbddb* ([6]).

Table 1 lists all eight benchmarks. For each benchmark, it shows the number of total methods (#methods), the number of application methods (#app_methods) and the number of total statements (#statements). All the numbers are calculated on the call graph generated by the *0-CFA* analysis using on-the-fly construction. The statements are *Soot*'s *jimple* statements, which is a three-address representation for Java bytecode. From this table it can be observed that a large number of library methods exist in a whole-program call graph: even in the smallest benchmark (*compress*) with only 60 application methods, the whole-program call graph still contains 3468 methods, 3408 of which are library methods.

### 4.2 Accuracy.

Table 2 shows the size of the generated application call graph in terms of the number of call edges. Each call edge is a four-tuple $<$ *caller*, *call site*, *type*, *callee* $>$, in which *type* can be either $direct$ or $callback$. Both $simple$ and $new$ algorithms generate the same set of $direct$ call edges, which correspond to the call edges between application methods in the whole-program call graph. The numbers of $direct$ call edges are shown in column *#direct*. The numbers of $callback$ edges generated by both algorithms are shown in the columns *#callback*. The reduction rate achieved by the $new$ algorithm over the $simple$ one is also shown.

Benchmark *compress* does not have $callback$ edges. Both algorithms produce precise results, so its reduction rate is unavailable and thus not counted in calculating the average. The $new$ algorithm reduces at least 43% of the $callback$ edges generated by the $simple$ one for all the other seven benchmarks. On average, the reduction rate is 74.94%, that amounts to an overall 64.43% on average call edge reduction for generated application call graphs. Note that benchmark *mtrt* is a dual-threaded version of *raytrace*, and the calculated $callback$ edges are exactly the same for both benchmarks by either algorithm, so they are regarded as one benchmark and only counted once in calculating the average.

| benchmark | *#direct* | *#callback* | | |
|---|---|---|---|---|
| | | *simple* | *new* | Reduction |
| compress | 122 | 0 | 0 | NA |
| jess | 2241 | 17790 | 10001 | 43.78% |
| raytrace | 1081 | 3400 | 129 | 96.21% |
| db | 158 | 5088 | 1455 | 71.40% |
| javac | 13069 | 43241 | 17889 | 58.63% |
| mpegaudio | 689 | 7659 | 29 | 99.62% |
| mtrt | 1082 | 3400 | 129 | 96.21% |
| jack | 1283 | 8076 | 1614 | 80.01% |
| | | | Average | 74.94% |

**Table 2. Generated Application Call Graph**

| benchmark | Time Cost(sec) | | #Library_Calls |
|---|---|---|---|
| | *0-CFA* | $V^a$-$DataReach^{ft}$ | |
| compress | 41 | 360 | 806 |
| jess | 46 | 468 | 1584 |
| raytrace | 43 | 446 | 986 |
| db | 41 | 417 | 994 |
| javac | 56 | 781 | 2432 |
| mpegaudio | 57 | 427 | 904 |
| mtrt | 43 | 445 | 986 |
| jack | 53 | 465 | 1884 |

**Table 3. Time Cost for $New$ Algorithm**

### 4.3 Practicality.

Table 3 shows the time cost for the $new$ algorithm. There are mainly two phases that cost considerable time: one is *0-CFA*, the whole-program points-to analysis, and the other is $V^a$-$DataReach^{ft}$ to resolve library callbacks. It can be seen that the algorithm is practical in that it finishes in reasonable time for all benchmarks.

$V^a$-$DataReach^{ft}$ performs a call-site specific points-to analysis for each library call, so its time cost is closely related to the number of library calls in the benchmark. In order to show the correlation between the time cost and the number of library calls, we chose various subsets of all library calls in benchmark *javac*, and applied $V^a$-$DataReach^{ft}$ to them. The subsets are chosen randomly and cumulatively. For example, initially we chose 100 library calls randomly as the first subset, then we chose another 100 randomly and added them to the first subset to form the second one. As shown in Figure 4, the $x$-axis is

| benchmark | #methods | #app_methods | #statements | Description |
|---|---|---|---|---|
| compress | 3468 | 60 | 20271 | A high-performance application to compress or uncompress large files; based on the Lempel-Ziv method(LZW) |
| jess | 3907 | 465 | 23163 | A Java expert shell system based on NASA's CLIPS system |
| raytrace | 3610 | 190 | 21541 | Ray tracer application |
| db | 3480 | 66 | 20555 | Performs database functions on a memory-resident database |
| javac | 4661 | 1155 | 27574 | JDK 1.0.2 Java compiler |
| mpegaudio | 2667 | 256 | 21215 | MPEG-3 audio file compression application |
| mtrt | 3610 | 190 | 21542 | Dual-threaded version of raytrace |
| jack | 3736 | 318 | 22854 | A Java parser generator with lexical analyzers (now JavaCC) |

**Table 1. Benchmarks Description**

the number of library calls in a chosen subset. The $y$-axis is the time cost to run $V^a$-$DataReach^{ft}$ on all library calls in a subset using our implementation. We can see that the time cost increases more slowly than the number of library calls (e.g., the size-100 subset costs 249 seconds, while the size-2000 subset costs 696 seconds, much less than 4980 (i.e., 249*2000/100) seconds). The reason is that a BDD-based solver is used to implement $V^a$-$DataReach^{ft}$, and several call-site specific points-to sets for different library calls can be updated at the same time with a single BDD operation, so the implementation is more scalable than performing those analyses one after another.

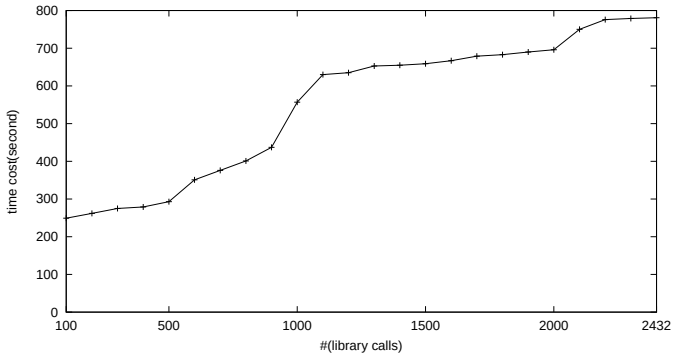



**Figure 4. Time Cost for $V^a$-$DataReach^{ft}$ on *javac* with Increasing Numbers of Library Calls**

# 5 Potential Usage Of Application Call Graph

An accurate application call graph is useful in many software engineering applications. Compared to the whole-program call graph, the application call graph has the following two advantages: (1) it can capture the calling relationships more accurately among application methods; (2) an application call graph contains fewer nodes, and those algorithms whose cost is closely correlated to the call graph size will be more efficient if the application call graph can be substituted for the whole-program call graph. Based on these advantages, this section explores the potential usage of an accurate application call graph in white box testing.

**White Box Testing**

White box testing, a widely used testing technique, is also called clear box testing, glass box testing or structural testing. The term *white box* indicates that testing is done with specific knowledge of the code to be executed. A test coverage criterion is generated according to the control-flow and/or data-flow information from the code, and one goal of white box testing is to improve the test coverage ratio. There are different kinds of white box testing techniques as classified by the coverage criterion used, such as call-chain based testing [10], def-use pair based testing, etc.

## 5.1 Call-Chain Based Testing.

A *call chain* corresponds to a directed path on a call graph. Call-chain based testing involves static analysis and dynamic analysis: static analysis computes a set of call chains that may be observed during runtime. As a conservative estimate, this set is used as the test coverage requirement. Dynamic analysis observes the run-time behaviour and calculates the call chain coverage achieved during test execution.

The time cost allowed for testing an application is often limited. Unlike application methods, the library methods are usually considered well tested, so testing is usually focused on covering those call chains made up of application methods. Because the application call graph can capture the calling relationships more accurately among application methods than the whole-program call graph, it can be used to generate a more accurate set of application-method call chains including callbacks. This means that fewer infeasible call chains will be included in the test coverage requirements, so the unnecessary cost of test data generation and manual code inspection can be reduced. For example, as shown in Figure 1, `App.appendA()` $\rightarrow$ `B.toString()` and `App.appendB()` $\rightarrow$ `A.toString()` may be generated by a whole-program call graph, while an accurate application call graph will consider those call chains infeasible and not include them in the test coverage requirements.

### 5.2 Def-Use Pair Based Testing.

*Def-use pair* (DU-pair) based testing is referred to as *all-uses* in a classic definition for a family of data flow testing criteria [9]; its goal is to cover all possible uses for each definition during test execution. Similar to call-chain based testing, DU-pair based testing involves a static analysis to calculate the set of DU-pairs as a test coverage requirement, and dynamic analysis to measure the achieved coverage. The static analysis requires a call graph to compute inter-procedural DU-pairs.

For a Java application, one problem for DU-pair based testing is that many DU-pairs exist in the library, so the set calculated statically may contain too many DU-pairs for the test execution to achieve a decent coverage ratio, which makes DU-pairs unrealistic to use as a test criterion. In addition to the use of the application call graph, our solution to this problem is to generate a *summary* statement for each library call from the application to summarize side effect information for this library entry. A *summary* statement consists of the following three kinds of operations:

I. Call an application method.

II. Read from an object field [2].

III. Write to an object field.

Each *summary* statement corresponds to a library call from the application program. The application methods called by a *summary* statement are the $callback$ targets of this library call, as calculated by $V^a$-$DataReach^{ft}$. We want to capture the *live* definitions and *live* uses for the library entry points, so the objects in II and III only include those that are accessible from the code executed beyond this library entry, namely:

1. objects that are initialized before the library call and passed in through parameter or instance field read statements.
2. objects that are accessible by another thread during the lifetime of the library call.
3. objects that are accessible to the code executed in a $callback$ target method or its descendants.
4. objects that are accessible through the return node of the library call to the code executed after the call finishes.

The set of the above objects is denoted by *AllAccessible*. There may be more objects whose fields are read or written during this library entry, but if they are not in *AllAccessible*, then any read from or write to them is regarded as a local operation and will not be summarized in the corresponding *summary* statement.

[2] As in the previous sections, static fields and arrays are omitted in the discussion.

*AllAccessible* is different from *Accessible* calculated by the $V^a$-$DataReach^{ft}$ algorithm in that *Accessible* only contains the objects that are accessible from the code executed beyond this library entry **before** the call finishes (i.e., cases 1-3). If the set of objects in case 4 is denoted as *LaterAccessible*, then *AllAccessible* is the union of *Accessible* and *LaterAccessible*.

After the $V^a$-$DataReach^{ft}$ algorithm finishes, *AllAccessible* can be calculated by the following constraints for a given library call, in which $U$ is the result for the call-site specific points to analysis, $M$ is a possible callee for the original call and $M.ret\_var$ is the reference variable returned by $M$:

$$\begin{cases} U_{M.ret\_var} \subseteq LaterAccessible \wedge \\ U_{o.f} \subseteq LaterAccessible \; \forall o \in LaterAccessible \wedge \\ LaterAccessible \subseteq AllAccessible \wedge \\ Accessible \subseteq AllAccessible \end{cases}$$

For a method called by the library call, if there is a reference variable returned by the method, the reference variable's $local$ points-to set is included in $LaterAccessible$. Also, all objects reachable via field references from the returned variable according to the call-site specific points-to result are included in $LaterAccessible$.

After $AllAccessible$ is computed, the object fields read from and written to by the $summary$ statement are calculated using the following constraints, and denoted by the sets *Read* and *Write* respectively:

(a). For each method $M$ in $R$, and for each instance field read statement $l = r.f$ in $M$ and each $o_i \in U_r$:
$(o_i \in AllAccessible) \Rightarrow o_i.f \in Read$

(b). For each method $M$ in $R$, and for each instance field write statement $w.f = l$ in $M$ and each $o_i \in U_w$:
$(o_i \in AllAccessible) \Rightarrow o_i.f \in Write$

(c). For each method $M$ in $R$, and for each static field read statement $l = C.f$ in $M$:
$C.f \in Read$

(d). For each method $M$ in $R$, and for each static field write statement $C.f = l$ in $M$:
$C.f \in Write$

Given a library call, the sets *Callback*, *Read* and *Write* can be generated by $V^a$-$DataReach^{ft}$ and the above calculation. A *summary* statement is assumed to perform the following operations: call methods in *Callback*, read from each object field in *Read* and write to each object field in *Write* [3]. By substituting the *summary* statement for each library call, the DU-pairs excluding those in the library can be calculated as test coverage requirement. Also, static analysis to calculate the DU-pairs can be performed on the application call graph, guaranteeing efficiency because of the fewer method nodes, and accuracy because of the spurious $callback$ edges eliminated.

[3] As a safe approximate, the writes by the *summary* statement are assumed to be *non-killing*. Flow-sensitive analysis is needed to further improve the precision.

## 6 Related Work

**Call Graph Construction & Reference Analysis**

Grove and Chambers presented a large number of call graph construction algorithms for object-oriented languages [4]. There is also a wide range of reference and points-to analyses [12, 5] that can be used to construct call graphs. The key contribution of our work is that we explore approaches to build application call graphs that can capture calling relationships among application methods induced by paths through the library more accurately than the whole-program call graphs built by the previous work.

**Data Reachability Algorithm**

The algorithm $V^a$-$DataReach$ presented in this paper is one variant of the data reachability algorithm presented in [3], in which the data reachability algorithm was used to statically discover Java exception throw-catch pairs accurately. The data reachability algorithm calculates the methods or sub-call graph reachable from a call site for object-oriented program. [3] presents a detailed discussion and three forms of data reachability algorithm: *DataReach*, *M-DataReach* and *V-DataReach*, listed in increasing order of precision. One key contribution of work is $V^a$-$DataReach$, that differs from *V-DataReach* in calculating the accessibility information on the fly, as discussed in Section 3.2.

There are also several other algorithms to detect infeasible control flow paths statically ([1, 16]). Their difference from the data reachability algorithm is discussed in [3].

## 7 Conclusion and Future Work

In this paper we have explored approaches to construct an accurate *application call graph* for Java. We designed a new variant of the data reachability algorithm and fine tuned it to resolve the library $callback$ edges accurately. The experimental study shows that the proposed new algorithm is practical and eliminates a large amount of spurious $callback$ edges from the application call graph generated by a simple algorithm: on average, the number of $callback$ edges is reduced by 74.97%, amounting to an overall on average 64.43% edge reduction for the generated application call graphs.

There are mainly two directions for our future work. One is to evaluate algorithm accuracy through more empirical studies; the other is to explore the use of accurate application call graphs in white box testing, to see its improvement in generating a better test coverage requirement.

## References


[1] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 361–377. Springer–Verlag, 1997.

[2] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.

[3] C. Fu, A. Milanova, B. G. Ryder, and D. Wonnacott. Robustness Testing of Java Server Applications. *IEEE Transactions on Software Engineering*, 31(4):292–311, Apr. 2005.

[4] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6), 2001.

[5] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE*, pages 54–61, 2001.

[6] http://bddbddb.sourceforge.net/. bddbddb:bdd-based deductive database.

[7] http://www.sable.mcgill.ca/soot/. Soot: a java optimization framework.

[8] O. Lhotak and L. Hendren. Scaling java points-to analysis using spark. *International Conference on Compiler Construction*, 2003.

[9] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, Apr. 1985.

[10] A. Rountev, S. Kagan, and M. Gibas. Static and dynamic analysis of call chains in Java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–11, 2004.

[11] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. In *Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications*, pages 43–55, 2001.

[12] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Proceedings of the Twelveth International Conference on Compiler Construction, pages 126–137, April 2003. invited paper.

[13] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.

[14] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[15] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[16] A. L. Souter and L. L. Pollock. Characterization and automatic identification of type infeasible call chains. *Information and Software Technology*, 44(13):721–732, October 2002.

[17] Specbench.org. Java client/server benchmarks.

[18] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications*, pages 281–293, Oct. 2000.

[19] W. Zhang and B. Ryder. A Semantics-Based Definition for Interclass Test Dependence. Technical Report DCS-TR-597, Department of Computer Science, Rutgers University, January 2006.