JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2007; **19**:231–252
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.351

Research

Automatic construction of accurate application call graph with library call abstraction for Java

Weilei Zhang[*,†] and Barbara G. Ryder

Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ, 08854, U.S.A.

SUMMARY

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 for generating more accurate application call graphs for Java. A new data reachability algorithm is proposed and fine tuned to resolve library callbacks accurately. Compared with an algorithm that resolves library callbacks by traversing the whole-program call graph, the fine-tuned data reachability algorithm results in fewer spurious callback edges. In empirical studies, the new algorithm shows a significant reduction in the number of spurious callback edges. On the basis of the new algorithm, a library abstraction can be calculated automatically and applied in amortized slicing and dataflow testing. Copyright © 2007 John Wiley & Sons, Ltd.

Received 29 January 2007; Revised 8 May 2007; Accepted 8 June 2007

KEY WORDS: call graph; library callback; Java analysis

1. INTRODUCTION

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.

[*]Correspondence to: Weilei Zhang, Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscat away, NJ, 08854, U.S.A.
[†]E-mail: weileiz@cs.rutgers.edu

In object-oriented languages, a call site may invoke several callees due to dynamic dispatch. The corresponding call graph construction [1] uses some form of *reference analysis*. Reference analysis calculates the type information about the objects to which reference variables can point during program execution. There is a wide variety of reference analyses that differ in terms of cost and precision. An in-depth discussion can be found in [2,3].

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 an application program if an accurate application call graph can be substituted for a whole-program call graph.

1.1. 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, $a \to m_1 \to m_2 \to \cdots \to m_n \to b$ 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.

With the appropriate library call abstraction, an accurate application call graph can be used in many software engineering applications more efficiently and effectively because it contains less nodes than a whole-program call graph. This benefit is more and more significant for modern object-oriented languages because the use of libraries is common and the libraries have grown larger. In Java, for example, since the first release the library size has increased from about two hundred to thousands of classes in a span of almost 10 years, with added support for collections, regular expressions, etc. Also, the library can be even larger in a multi-tier, framework-based software architecture: when the upper tier is the analysis focus, all the lower tiers can be considered as a part of library.

The contributions of this work are:

- Design of new approaches to construct an accurate *application call graph* for Java. An algorithm (V^a*-DataReach*) is proposed and fine tuned to resolve library callbacks accurately. The algorithm is a new variant of data reachability algorithm [4] in that it calculates escape information on the fly for each call site, rather than perform a separate escape analysis.
- Implementation of the proposed algorithm and experiments with it. As an extension to [5], this paper does extensive empirical studies, including the comparison between the new algorithm and points-to analyses with different context sensitivities.
- A complete description of an automatic calculation for library call abstraction and proposals for its usage in program slicing and dataflow testing.

1.2. 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 several uses of the accurate application call graph with library call abstraction. 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; hence, the precision for direct edges corresponds directly to the precision for the whole-program call graph [1]. Callback edges are generated by collapsing through-library call paths that connect a pair of application methods in the whole-program call graph. Many through-library call paths cannot happen at runtime (i.e., they are infeasible); 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` 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; hence, `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. However, 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.

```
class App{
  StringBuffer local;
  StringBuffer appendA(){
    A a=new A();
    return(local.append(a));
  }
   StringBuffer appendB(){
     B b=new B();
     return(local.append(b));
   }
 }
class A{
  String toString(){...}
}
 class B{
   String toString(){...}
 }
```

(a)

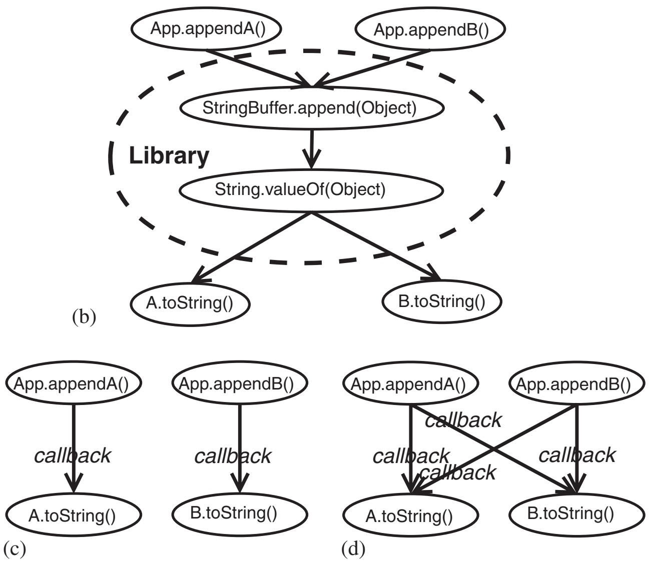


Figure 1. An example to illustrate the simple algorithm and its imprecision: (a) code; (b) whole-program call graph; (c) precise application call graph; and (d) application call graph generated by the simple algorithm.

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 [6,7], a whole-program context-sensitive analysis, but 2-CFA is very expensive. What is 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 [8] 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 [4] 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 with 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()` → `StringBuffer.append (Object)` → `String.valueOf(Object)` → `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 [4]: *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 set of 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). In V*-DataReach*, in order to calculate the set of those *accessible* objects, a global *escape* analysis [9] 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*, which 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

Similar 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 the set of *accessible* objects (*Accessible*), the call-site specific points-to result (U, $U \subseteq Pt$) 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 ($\mathscr{P}(O)$) 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. During the call-site specific analysis to calculate U, the points-to information for the fields of *accessible* objects comes from Pt, while the points-to information for the local reference variables and the fields of *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_is such that $a.f_1.f_2 \ldots\ldots f_n$ 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).

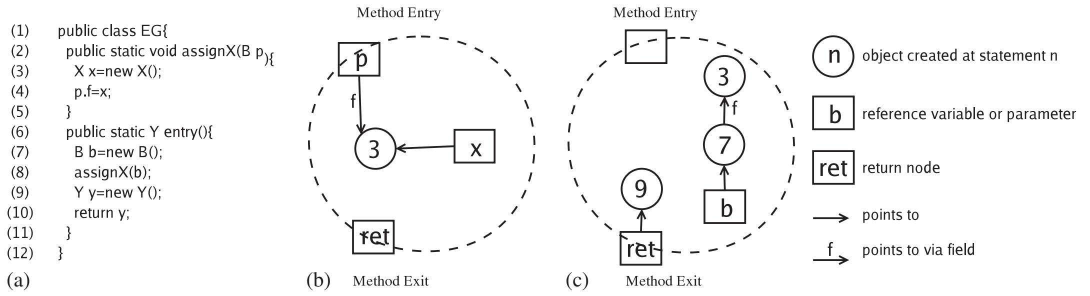
V^a-DataReach is defined by the following constraints, using the constraint-based formalism from [10], analogous to the data reachability algorithm schema defined in [4]:

- **input:**
$$\begin{cases} Pt: Ref \rightarrow \mathscr{P}(O),\ O \times F \rightarrow \mathscr{P}(O) \\ \text{the original call site as the starting point.} \end{cases}$$
- **output:**
$$\begin{cases} R \\ Accessible \\ U: Ref \rightarrow \mathscr{P}(O),\ O \times F \rightarrow \mathscr{P}(O) \end{cases}$$
- **initialize:** for each target M at original call and the corresponding actuals a_i and formals $M.p_i$:
$$\begin{cases} M \in R \wedge \\ Pt(a_i) \subseteq Accessible \wedge \\ Pt(a_i) \subseteq U_{M.p_i} \end{cases}$$
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 $l = r$ 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 $StaticLookup(o, m) = M'$:
$(M \in R) \wedge (o \in U_e) \Rightarrow$
$$\begin{cases} M' \in R \wedge \\ U_{e_i} \subseteq U_{M'.p_i} \text{ where } e_i \text{ is of reference type and } p_i \text{ is its corresponding formal} \\ \quad \text{parameter in } M' \wedge \\ U_{M'.ret\_var} \subseteq U_l \wedge \\ o \in U_{M'.this} \end{cases}$$
6. For each method M and for each static field write statement $C.f = r$ in M:
$(M \in R) \Rightarrow U_r \subseteq Accessible$
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:
$(M \in R) \Rightarrow$
$$\begin{cases} M' \in R \wedge \\ U_{e_i} \subseteq U_{p_i} \text{ where } e_i \text{ is of reference type and } p_i \text{ is its corresponding formal} \\ \quad \text{parameter in } M' \wedge \\ U_{M'.ret\_var} \subseteq U_l \end{cases}$$

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



```
(1)    public class EG{
(2)      public static void assignX(B p){
(3)        X x=new X();
(4)        p.f=x;
(5)      }
(6)      public static Y entry(){
(7)        B b=new B();
(8)        assignX(b);
(9)        Y y=new Y();
(10)       return y;
(11)     }
(12)   }
```

Figure 2. An example to illustrate the difference between V^a-DataReach and V-DataReach: (a) code; (b) `EG.assignX(B)`; and (c) `EG.entry()`.

(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 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 a virtual call, given the receiver object and the compile-time target method. Constraints 6–8 handle static field writes, static field reads and static call sites, respectively.

3.2.1. 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) and (c) illustrate the points-to graphs 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. Hence, 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*; hence, 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*. Hence, *V^a-DataReach* can obtain more accurate results than *V-DataReach*.

3.2.2. *Using V^a-DataReach to resolve library callbacks*

If in constraints 5 and 8 of V^a-*DataReach*, the reachable call edge $\langle M, cs, M'\rangle$ is recorded for each method M' reached from 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 the accuracy, as illustrated in Figure 3.

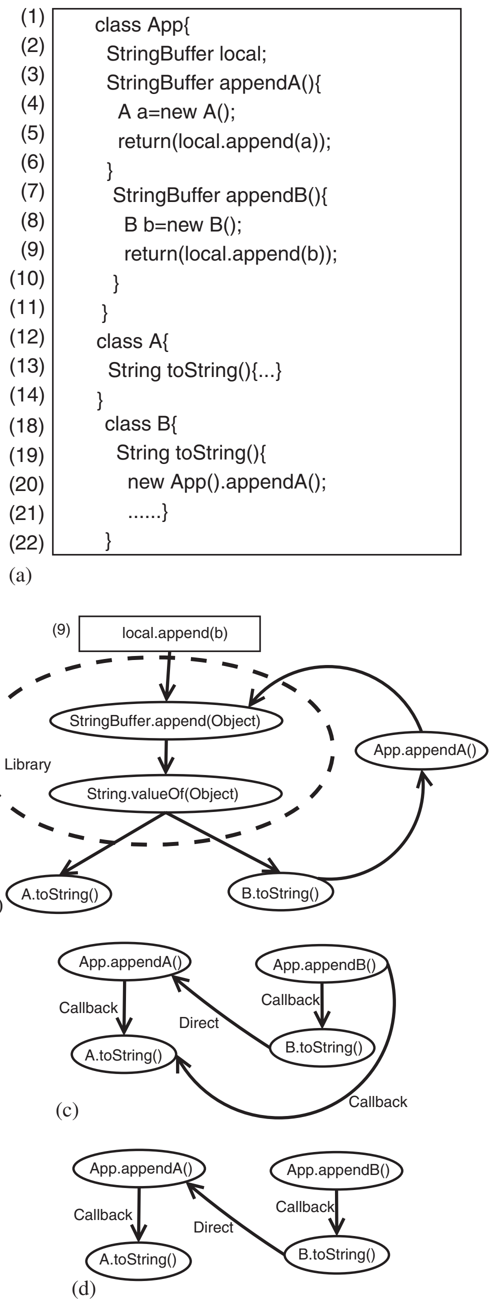
Figure 3(a) is a slight modification of Figure 1(a), where method `B.toString()` contains one more statement in line 20. Originally in Figure 1, V^a-*DataReach* determines that call site (9) of method `App.appendB()` calls back `B.toString()` only. However, the new codes in Figure 3(a) introduce the 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 with 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 $M \in \mathit{Lib}$, for each virtual call site $l = e.m(e_1, \ldots, e_n)$ occurring in M, and for each $o \in Pt(e)$ where $\mathit{StaticLookup}(o, m) = M'$ and p_i are the formal parameters of M': $(M \in R) \wedge (o \in U_e) \Rightarrow$

$$\left\{\begin{array}{l} M' \in \mathit{Lib} \Rightarrow \left\{\begin{array}{l} M' \in R\, \wedge \\ U_{e_i} \subseteq U_{M'.p_i} \wedge \\ U_{M'.ret\_var} \subseteq U_l \wedge \\ o \in U_{M'.this} \end{array}\right. \\ M' \notin \mathit{Lib} \Rightarrow \left\{\begin{array}{l} M' \in \mathit{Callback} \wedge \\ U_{e_i} \subseteq \mathit{Accessible} \wedge \\ Pt(M'.ret\_var) \subseteq U_l \wedge \\ Pt(M'.ret\_var) \subseteq \mathit{Accessible} \end{array}\right. \end{array}\right.$$

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



```
(1)  class App{
(2)   StringBuffer local;
(3)   StringBuffer appendA(){
(4)    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{
(19)   String toString(){
(20)    new App().appendA();
(21)    ......}
(22)  }
```

Figure 3. An example to illustrate the need for fine-tuned algorithm: (a) code for B.toString(); (b) discovered sub call graph while running general data reachability algorithm on call site (9) in (a); (c) application call graph generated by V^a-DataReach; and (d) actual application call graph.

library call finishes. There is also a similar substitute for constraint 8 in V^a*-DataReach* to handle static call sites:

8. For each method $m \in Lib$, and for each static call site $l = C.M'(e_1, \ldots, e_n)$ in m where p_i are the formal parameters of M':
$(m \in r) \rightarrow$

$$\begin{cases} M' \in Lib \rightarrow \begin{cases} M' \in r \wedge \\ U_{e_i} \subseteq U_{M'.p_i} \wedge \\ U_{M'.ret\_var} \subseteq U_l \end{cases} \\ \\ M' \notin Lib \rightarrow \begin{cases} M' \in callback \wedge \\ U_{e_i} \subseteq accessible \wedge \\ pt(M'.ret\_var) \subseteq U_l \wedge \\ pt(M'.ret\_var) \subseteq accessible \end{cases} \end{cases}$$

3.4. Complexity for algorithm V^a-DataReach^{ft}

Given the result of a points-to analysis, algorithm V^a*-DataReach*^{ft} resolves the callbacks from a specific library call according to the eight constraints. Each constraint corresponds to a statement related to reference variables or fields (e.g., constraint 1 corresponds to the object creation statement and constraint 2 corresponds to the reference assignment statement, etc.). Given a program, let $\mathscr{S}^l_i$ $(1 \le i \le 8)$ represent the number of statements corresponding to constraint i in the library, and $\mathscr{S}^l$ represent the total number of those statements (i.e., $\mathscr{S}^l = \sum_{i=1}^{8} \mathscr{S}^l_i$). Also, let $\mathscr{LC}$ be the number of library calls from application methods, and $\mathscr{O}$ be the number of object creation sites in the whole program.

The constraints are defined recursively and a fixed-point calculation is performed to solve the constraints. For example, a change in the call-site specific points-to result for a reference variable (i.e., U_{ref}) at constraints 1, 2, 4, 5, 7 and 8 causes recalculation of constraints 3, 4, 5, 6 and 8; similarly a change in the call-site specific points-to result for an object field (i.e., $U_{o.f}$) at constraint 3 causes recalculation of constraint 4. During the recalculation of a constraint such as $X \subseteq Y$ in which X and Y are two sets, only the newly added elements in X are inserted into Y; it cost $\mathscr{O}(1)$ to insert one element into a set. Hence, the time cost for one such constraint is bounded above by the size of X. Similarly, the time cost for each constraint is broken down as follows:

- Constraint 1 is calculated once for each object creation statement in the reachable library methods; hence, its time cost is $O(\mathscr{S}^l_1) \times O(1) = O(\mathscr{S}^l_1)$.
- For each reference assignment statements $l = r$, constraint 2 is recalculated when the points-to result U_r changes. The size of U_r is at most $\mathscr{O}$, and there are at most $\mathscr{S}^l_2$ reference assignment statements in the reachable library methods; hence, the total time cost for constraint 2 is $O(\mathscr{S}^l_2 \times \mathscr{O})$.
- For each instance field write statement $l.f = r$ in the reachable library methods, constraint 3 is recalculated when the points-to result U_r or U_l changes; hence, its time cost is $O(\mathscr{S}^l_3 \times \mathscr{O}^2)$.
- For each instance field read statement $l = r.f$ in the reachable library methods, constraint 4 is recalculated when the points-to result U_r or $U_{o_i.f}$ $(o_i \in U_r)$ changes; hence, its time cost is $O(\mathscr{S}^l_4 \times \mathscr{O}^2)$.

- For each virtual call site $l = e.m(e_1, \ldots, e_n)$ in the reachable library methods, the calculation for constraint 5 is divided up into two phases: (i) when U_e changes, the *StaticLookup* function is checked to see if a new callee method will be resolved, and if so, $O(n)$ inclusion constraints are added between the points-to results for each actual reference parameter and its corresponding formal parameter (i.e., $U_{e_i} \subseteq U_{M'.p_i}$); (ii) for each of the above-mentioned inclusion constraints, any change in U_{e_i} causes the constraint to be recalculated. The time cost for phase (i) is $O(\mathscr{S}_5^l \times \mathscr{O} \times n)$. Let t be the number of possible callee targets for each virtual call site, then $O(t \times n)$ inclusion constraints can be generated for each call site. Hence, the time cost for phase (ii) is $O(\mathscr{S}_5^l \times \mathscr{O} \times t \times n)$, which dominates $O(\mathscr{S}_5^l \times \mathscr{O} \times n)$. It can be shown that $t \leq \mathscr{O}$; hence, the total time cost for constraint 5 is $O(\mathscr{S}_5^l \times \mathscr{O}^2 \times n)$.
- For each static field write statement $C.f = r$ in the reachable library methods, constraint 6 is recalculated when the points-to result U_r changes. Hence, its time cost is $O(\mathscr{S}_6^l \times \mathscr{O})$.
- For each static field read statement $l = C.f$ in the reachable library methods, constraint 7 is calculated at most once to insert all elements in $U_{C.f}$ into U_l. Hence, its time cost is $O(\mathscr{S}_7^l \times \mathscr{O})$.
- For each static call site in the reachable library methods, the calculation for constraint 8 is divided up into two phases, similar to that for constraint 5. The difference between a static call site and a virtual one is that it has exactly one possible callee (i.e., $t = 1$); hence, the time cost for constraint 8 is $O(\mathscr{S}_8^l \times \mathscr{O} \times n)$.

To sum up, given a library call, the time cost For V^a-*DataReach*^{ft} to resolve its callbacks is $O(\mathscr{S}_1^l + \mathscr{S}_2^l \times \mathscr{O} + \mathscr{S}_3^l \times \mathscr{O}^2 + \mathscr{S}_4^l \times \mathscr{O}^2 + \mathscr{S}_5^l \times \mathscr{O}^2 \times n + \mathscr{S}_6^l \times \mathscr{O} + \mathscr{S}_7^l \times \mathscr{O} + \mathscr{S}_8^l \times \mathscr{O} \times n)$. Usually the number of parameters for a method is small and not dependent on the program size; hence, we can consider n to be a constant. Also because $\mathscr{S}^l = \sum_{i=1}^{8} \mathscr{S}_i^l$, the time cost to resolve callbacks is $O(\mathscr{S}^l \times \mathscr{O}^2)$ for one library call, and $O(\mathscr{L}\mathscr{C} \times \mathscr{S}^l \times \mathscr{O}^2)$ for the whole program.

4. EMPIRICAL STUDY

We implemented the algorithm V^a-*DataReach*^{ft} presented in Section 3 to resolve callback edges and generate the accurate application call graph. This section describes our experiments with the algorithm compared with the simple algorithm presented in Section 2 that resolves callbacks by traversing the whole-program call graph. The simple algorithm is parameterized by the points-to analysis. The callback edges can be resolved more accurately as the precision for the points-to analysis increases with more context information; thus, we want to perform further comparison between V^a-*DataReach*^{ft} and well-known context-sensitive points-to analyses in terms of callback resolution. V^a-*DataReach*^{ft} is also parameterized by the whole-program points-to analysis, and we want to see the effect of context-sensitive whole-program analyses on the precision of V^a-*DataReach*^{ft}.

In summary, our goal is to answer the following three questions during the empirical study:

- Parameterized with the same context-insensitive points-to analysis (*0-CFA* [11,12]), how many spurious callback edges can be eliminated by V^a-*DataReach*^{ft} from those generated by the simple algorithm?

- How many spurious callback edges can be eliminated by V^a-*DataReach*^{ft} compared with using a fully context-sensitive points-to analyses (i.e., V^a-*DataReach*^{ft} parameterized with context-insensitive points-to analysis vs the simple algorithm parameterized with context-sensitive ones)?
- How much more improvement can be achieved by V^a-*DataReach*^{ft} if it is parameterized by a context-sensitive points-to analysis rather than by a context-insensitive one?

4.1. Experiment setup

We experimented on 10 benchmarks, including all eight in the SPEC jvm98 suite [13], SPEC JBB2000[‡] and muffin[§]. All experiments were run on a 1.8 GHz AMD Athlon(tm) 64 Processor 3000+, 2 GB-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 [14], which utilizes a Java optimization framework, *Soot* [15] and a BDD-based constraint solver, *bddbddb* [16].

Note that benchmark *compress* does not have callback edges, and all algorithms produce this precise result; therefore, this benchmark is not listed. Benchmark *mtrt* is a dual-threaded version of *raytrace*, and the calculated callback edges are exactly the same for both benchmarks; hence, they are reported here as one benchmark. Table I lists the other eight benchmarks. For each benchmark, it shows the total number of classes (#classes), the total number of methods (#methods), the number of application methods (#app_methods), the number of statements (#statements), the number of creation sites (#creation_sites) and the number of library calls from the application program (#library_calls). All the numbers are calculated on the call graph generated by the *0-CFA* analysis using on-the-fly construction. The number of statements counted are statements in *jimple*, the three-address representation of Java bytecode used in *Soot*. From this table, it can be observed that a large number of library methods exist in a whole-program call graph. Taking benchmark *db* as an example, there are only 66 application methods, but the whole-program call graph contains 3480 methods, 3414 of which are library methods.

4.2. Precision: V^a-*DR*^{ft}*_0CFA* vs ***Simple_0CFA***

Table II shows the size of the generated application call graph in terms of the number of call edges. Each call edge is a four-tuple $\langle$*caller*, *call site*, *type*, *callee*$\rangle$, in which *type* can be either direct or callback. The numbers of the resolved callback edges are shown in the columns *#callback*. The callback edges are resolved by the simple algorithm and V^a-*DataReach*^{ft}. Both algorithms depend on and are parameterized by a whole-program points-to analysis. For comparison, the same *0-CFA* analysis, a widely used points-to analysis, is used for both algorithms. The simple algorithm and V^a-*DataReach*^{ft} parameterized by *0-CFA* are denoted as *Simple_0CFA* and V^a-*DR*^{ft}*_0CFA*, respectively, in the table. Let *#S0CFA* and *#DR*0 denote the numbers of callback edges calculated by the two algorithms, respectively. Then the reduction rate achieved by V^a-*DR*^{ft}*_0CFA* over *Simple_0CFA* is calculated as $(1 - \#DR0 \div \#S0CFA) \times 100\%$, and the data are shown in column *Reduction*. The same *0-CFA* points-to analysis results are used to generate the direct call edges, which correspond

[‡]SPEC's first Java Business Benchmark, http://www.spec.org/jbb2000/.
[§]An open-source WWW filtering system, http://muffin.doit.org/.

Table I. Benchmarks description.

Benchmark	#classes	#methods	#app_methods	#statements	#creation_sites	#library_calls
jess	757	3907	465	23 163	4060	1584
raytrace(mtrt)	673	3610	190	21 541	3697	986
db	646	3480	66	20 555	3611	994
javac	832	4661	1155	27 574	4487	2432
mpegaudio	694	2667	256	21 215	4605	904
jack	699	3736	318	22 854	3892	1884
jbb	878	4794	360	30 054	4348	2508
muffin	1171	6543	639	36 249	5820	2991

Table II. Generated application call graph.

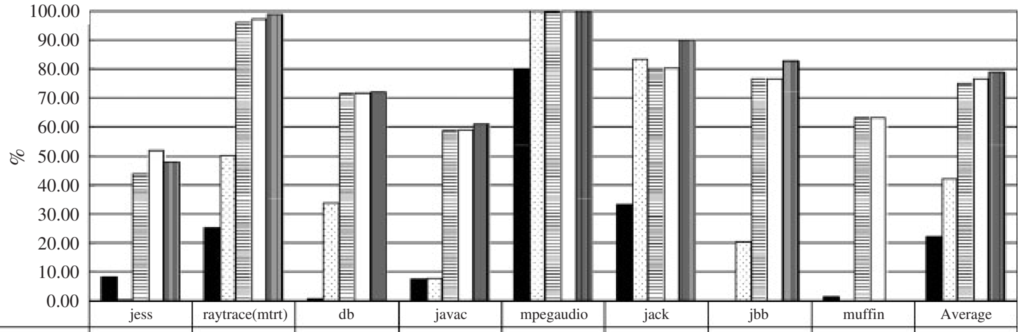
		#callback		
Benchmark	*#direct*	*Simple_0CFA*	V^a-*DR*^{ft}*_0CFA*	*Reduction* (%)
jess	2241	17 790	10 001	43.78
raytrace(mtrt)	1081	3400	129	96.21
db	158	5088	1455	71.40
javac	13 069	43 241	17 889	58.63
mpegaudio	689	7659	29	99.62
jack	1283	8076	1614	80.01
jbb	1457	8786	2046	76.71
muffin	1894	31 419	11 583	63.13
			Average	73.69

to the call edges between application methods in the whole-program call graph. The number of direct call edges is shown in column *#direct*.

V^a-*DR*^{ft}*_0CFA* reduces at least 43% of the callback edges generated by *Simple_0CFA* for all eight benchmarks. On average, the reduction rate is 73.69%; this amounts to an overall 63.99% on average call edge reduction for the generated application call graphs. Benchmark *jess* is studied further to understand algorithm imprecision: a lot of spurious callbacks found by V^a-*DataReach*^{ft} are due to the extensive use of *java.util.Hashtable*. *Hashtable* elements are internally stored in an array, which is regarded as one allocation site in our implementation. Different *Hashtable* instances share the same allocation site for the array, and thus, their points-to information passing through it will be merged. V^a-*DataReach*^{ft} cannot solve the problem because the array is not a *local* object for the *call-site-specific* points-to analysis, and its points-to information comes from the imprecise global points-to analysis. A more accurate data reachability algorithm with a more precise naming scheme for objects (e.g., a context-sensitive naming scheme to distinguish the different objects created at the same allocation site [17,18]) will solve the problem, but the above-mentioned naming scheme induces both space and time cost, and we have not been able to make this approach scalable.

4.3. Context-sensitive points-to analysis

In this subsection, we compare V^a-*DR*^{ft}*_0CFA* with the simple algorithm parameterized by different context-sensitive points-to analyses; we also evaluate the effect of context-sensitive points-to analyses on the precision of V^a-*DataReach*^{ft}.



	jess	raytrace(mtrt)	db	javac	mpegaudio	jack	jbb	muffin	Average
■ Simple_1CFA	8.40	25.18	0.47	7.58	80.08	33.33	0.11	1.40	22.16
□ Simple_1OB	0.17	50.18	33.73	7.64	99.95	83.41	20.23	N/A	42.18
▤ Va-DRft_0CFA	43.78	96.21	71.40	58.63	99.62	80.01	76.71	63.13	75.20
□ Va-DRft_1CFA	51.81	97.15	71.88	59.06	99.95	80.26	76.85	63.28	76.71
▥ Va-DRft_1OB	47.79	99.03	71.99	60.94	99.95	90.13	82.87	N/A	78.96

Figure 4. Callback resolution reduction rate achieved by *Simple_1CFA*, *Simple_1OB*, V^a*-DR*^{ft}*_0CFA*, V^a*-DR*^{ft}*_1CFA* and V^a*-DR*^{ft}*_1OB*, respectively, over *Simple_0CFA* (*muffin not included in the average calculation*).

There are two main kinds of context-sensitive points-to analyses: *n-CFA* [7] distinguishes contexts by the most recent *n* call sites, and n-object-sensitive analysis (denoted as *n-OB* [17]) distinguishes contexts by the receiver objects on which the most recent *n* methods are invoked. We implemented a *1-CFA* and a *1-OB* points-to analysis. The simple algorithms and V^a*-DataReach*^{ft} parameterized by both points-to analyses are denoted as *Simple_1CFA*, *Simple_1OB*, V^a*-DR*^{ft}*_1CFA* and V^a*-DR*^{ft}*_1OB*, respectively. Figure 4 shows the reduction rate in resolved library callbacks achieved by the above-mentioned four and V^a*-DR*^{ft}*_0CFA* over *Simple_0CFA*.

Note that *1-OB* points-to analysis runs out of memory and does not scale on benchmark *muffin* in our current implementation configuration; therefore, the corresponding *Simple_1OB* and V^a*-DR*^{ft}*_1OB* data are not available. We did not include *muffin* in any average calculated in this subsection. The exact reason for this lack of scalability is under investigation. However, we hypothesize that two factors may lead to this problem: (i) from Table I, it can be seen that among all eight benchmarks, *muffin* contains the largest number of classes, methods, statements and creation sites; (ii) a lot of methods in package *java.io* are called in *muffin*, and the complicated and highly polymorphic program structure in the java IO library makes *1-OB* hard to scale.

Examining the first three rows of Figure 4 to compare V^a*-DR*^{ft}*_0CFA* with *Simple_1CFA* and *Simple_1OB*, we see that V^a*-DR*^{ft}*_0CFA* usually eliminates more spurious library callbacks. On average, the reduction rate achieved by V^a*-DR*^{ft}*_0CFA* over *Simple_0CFA* is 75.20%, significantly better than 22.16% for *Simple_1CFA* and 42.18% for *Simple_1OB*. V^a*-DataReach*^{ft} can eliminate more spurious callbacks if parameterized with context-sensitive points-to analysis, as shown in the last three rows of Figure 4. The improvement is modest in that the average reduction rate improves from 75.20 to 76.71 and 78.96%.

The result for V^a*-DR*^{ft}*_0CFA* can be improved directly by performing intersection with the result for *Simple_1CFA* or *Simple_1OB*, instead of by running V^a*-DataReach* parameterized with context-sensitive points-to analysis. V^a*-DR*^{ft}*_0CFA*, *Simple_1CFA* and *Simple_1OB* are all approximate and

Table III. Break down callback edge sets calculated By *Simple_1CFA* (*S1CFA*), *Simple_1OB* (*S1OB*) and V^a-*DR*^{ft}*_0CFA* (*DR*0) (*muffin not included in the average calculation*).

	S1CFA	*S1OB*	*DR*0	*S1CFA* ∩ *DR*0		*S1OB* ∩ *DR*0		*S1OB* ∩ *S1CFA*	
Benchmark	#	#	#	#	Reduction (%)	#	Reduction (%)	#	Reduction (%)
jess	16 296	17 760	10 001	9286	47.80	10 001	43.78	16 296	8.40
raytrace(mtrt)	2544	1694	129	97	97.15	65	98.09	1694	50.18
db	5064	3372	1455	1455	71.40	1443	71.64	3372	33.73
javac	39 963	39 939	17 889	17 784	58.87	17 784	58.87	39 939	7.63
mpegaudio	1526	4	29	4	99.95	4	99.95	4	99.95
jack	5384	1340	1614	1610	80.06	797	90.13	1340	83.41
jbb	8776	7009	2046	2038	76.80	1520	82.70	7009	20.23
muffin	30 979	N/A	11 583	11 562	63.20	N/A	N/A	N/A	N/A
Average (%)					76.01		77.88		43.36

Table IV. Time cost.

Benchmark	*0-CFA*	*1-CFA*	*1-OB*	V^a-*DR*^{ft}*_0CFA*	V^a-*DR*^{ft}*_1CFA*	V^a-*DR*^{ft}*_1OB*
jess	46	206	280	468	653	872
raytrace	43	214	291	446	588	604
db	41	207	281	417	523	576
javac	56	408	505	781	933	1686
mpegaudio	57	222	306	427	474	537
jack	53	242	315	465	590	994
jbb	56	265	455	916	1241	2138
muffin	127	398	N/A	1714	2223	N/A

may result in different spurious callback edges. Table III breaks down the callback edge sets calculated by the three algorithms, which are abbreviated as *S1CFA*, *S1OB* and *DR*0 in the table to save space. The size of each set and set intersection (between each pair of sets) is shown in columns *#*. The difference between the callback edge sets calculated by different algorithms can be calculated as $S_a - S_b = S_a - S_a \cap S_b$. Take benchmark *raytrace* as an example. V^a-*DR*^{ft}*_0CFA* calculates much smaller callback edges sets (129) than the other two (2544 and 1694), but by calculating $129-97=32>0$ and $129-65=64>0$, we find the number of spurious callback edges that can be reduced by intersecting V^a-*DR*^{ft}*_0CFA* with each of the context-sensitive algorithms, respectively. The reduction rate achieved by direct intersection over *Simple_0CFA* is shown in columns *Reduction*. Compared with V^a-*DataReach* parameterized with context-sensitive points-to analysis, the improvement achieved by direct intersection is relatively less: 76.01 and 77.88% vs 76.71 and 78.96%, respectively, but at lower time cost, as shown in Table IV.

Table IV shows the time cost for different analyses. The result for a points-to analysis is required by V^a-*DR*^{ft}*_0CFA*, V^a-*DR*^{ft}*_1CFA* and V^a-*DR*^{ft}*_1OB*; hence the total time cost for them to resolve library callbacks is the sum of the time costs for the corresponding points-to analysis and the data reachability algorithm; for example, it costs 514 $(46+468)$ seconds for V^a-*DR*^{ft}*_0CFA* to calculate benchmark *jess*'s library callbacks, 859 (206+653) seconds for V^a-*DR*^{ft}*_1CFA* and 1152 (280+872) seconds for V^a-*DR*^{ft}*_1OB*. The time cost to get the intersection result between V^a-*DR*^{ft}*_0CFA* and *Simple_1CFA* (or *Simple_1OB*) is the sum of both costs, e.g., it is 674 (206 + 468)[¶] seconds for benchmark *jess*.

5. USAGE OF ACCURATE APPLICATION CALL GRAPH

An accurate application call graph is useful in many software engineering applications. Compared with the whole-program call graph, the accurate application call graph has the following two advantages: (1) it 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; (2) as shown in the experiments, it can capture the calling relationships more accurately among application methods. Because of these advantages, the accurate application call graph can be used effectively in those applications that rely on precise call graph construction. For example, it can be applied directly in call-chain-based testing [19]. Also, the proposed algorithm V^a-*DataReach*^{ft} can be easily extended to capture side-effect information for each library call; thus, by summarizing both callback and side-effect information, an *abstraction* statement can be generated to replace each library call in dependence-based analysis and applications, such as program slicing and dataflow testing. This section starts by illustrating how to generate an abstraction statement to summarize each library call in an accurate application call graph and then further explores its usage in program slicing and dataflow testing.

5.1. Library call abstraction

An abstraction statement is generated to summarize both the callback and side-effect information for each library call from the application program. It consists of the following three kinds of operations:

I Call an application method.
II Read from an object field[‖].
III Write to an object field.

The application methods called by an abstraction 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; hence, 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.

[¶]Actually extra time is needed: (i) the reachability algorithm on the call graph is performed by the simple algorithm to calculate callbacks and (ii) the intersection operation; but both are relatively cheap and can finish in seconds. Hence, they are omitted in the time-cost table for brevity.

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

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 abstraction statement.

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:

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

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 abstraction 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 $o_i \in U_r$:
$(o_i \in \mathit{AllAccessible}) \Rightarrow o_i.f \in \mathit{Read}$
(b) For each method M in R, and for each instance field write statement $w.f = l$ in M and each $o_i \in U_w$:
$(o_i \in \mathit{AllAccessible}) \Rightarrow o_i.f \in \mathit{Write}$
(c) For each method M in R, and for each static field read statement $l = C.f$ in M:
$C.f \in \mathit{Read}$
(d) For each method M in R, and for each static field write statement $C.f = l$ in M:
$C.f \in \mathit{Write}$

Given a library call, the sets *Callback*, *Read* and *Write* can be generated by V^a-*DataReach*^{ft} and the above calculation. An abstraction 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*[**].

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

5.2. Program slicing

A program *slice* consists of the set of program statements that potentially affect (*backward* slicing) or are affected by (*forward* slicing) the values computed at some point of interest, referred to as a *slicing criterion*. Program slicing was originally presented by Weiser in 1979 [20]. It has been widely applied in debugging, program comprehension, etc. A key data structure in program slicing is *SDG* (system dependence graph), introduced by Horwitz, Reps and Binkley for inter-procedural slicing [21]. The SDG is constructed from the (intra-procedural) program dependence graph and the call graph, and a two-pass traversal algorithm is performed on SDG to compute program slices.

Because of the large size of Java library, SDG construction is hard to scale on the whole-program call graph [22]. Our solution is *amortized* slicing, in which the construction of the SDG is performed in the following three steps:

I Construct an accurate application call graph with the algorithm V^a*-DataReach*^{ft}.
II Construct an abstraction statement for each library call according to the calculation in Section 5.1.
III Construct the SDG on the accurate application call graph with abstraction statements.

The essence of amortized slicing is to spend extra time in call graph construction and library call abstraction before constructing the SDG, to make the SDG construction more scalable. Also, this cost occurs only once for each program; and it can be amortized if multiple slices are calculated for one program.

One issue for amortized slicing is that the library codes will not be included in the calculated slices, but this is often desirable.

Debugging: A bug usually exists in a program statement that can influence the point where an error happens, and the developer can use backward slicing to expedite the debugging process by limiting the scope to search for the faulty statements. Because it consists of commonly used subroutines, the library is often considered to be fault free. In such cases, a whole-program slice is not so interesting as one restricted to application methods.

Program comprehension: When developers need to re-engineer or reverse-engineer an existing code base that they did not author, slicing can help them understand the program by discovering the flow of data and program dependences between code units. It is often the case that the developers already know the library very well; thus, they will be more interested in learning about the relationships between code units in the application program than those in the library. As for a multi-tier software architecture, sometimes developers may not know about the lower tiers, but usually they need to focus only on the tier on which they are working. Thus, the program statements in the lower tiers need not be included in the slices in order to understand the functionalities of the upper tier, and those lower tiers can be regarded similarly to a library during calculation.

5.3. Dataflow testing

Dataflow testing is a classic *white box* testing technique. The term white box indicates that testing is performed with specific knowledge of the code to be executed. A test coverage criterion is generated according to the control-flow and dataflow information from the code, and one testing goal is to improve the test coverage ratio. Different kinds of dataflow testing techniques are classified by the coverage criterion used [23], such as Def-Use(DU)-pair-based testing, sometimes referred to as *all-uses* testing.

The goal of DU-pair-based testing is to cover all possible uses for each definition during test suite execution. Similar to other dataflow testing techniques, DU-pair-based testing involves a static analysis for calculating the set of DU pairs as a test coverage requirement and dynamic analysis for measuring the achieved coverage. The static analysis requires a call graph to compute interprocedural DU pairs.

For a Java application, one problem for DU-pair-based testing is that many DU pairs exist in the library; hence, the set calculated statically may contain too many DU pairs for the test execution to achieve a decent coverage ratio, which makes the test criterion unrealistic. The time cost allowed for testing an application is often limited and the library is usually considered well tested; hence, testing is often focused on the application code. By substituting the abstraction statement for each library call, the application DU pairs excluding those in the library can be calculated as the test coverage requirement. The static analysis for calculating the DU pairs can be performed on the accurate application call graph, guaranteeing efficiency gained from fewer method nodes, and accuracy gained from eliminated spurious callback edges.

6. RELATED WORK

Several areas of research are closely related to this paper, including call graph construction, reference analysis, data reachability algorithm and infeasible path detection.

6.1. Call graph construction and reference analysis

Grove and Chambers presented a large number of call graph construction algorithms for object-oriented languages [1]. There is also a wide range of reference and points-to analyses [2,3] 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.

6.2. Data reachability algorithm and infeasible path detection

The algorithm V^a*-DataReach* presented in this paper is one variant of the data reachability algorithm presented in [4], 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. Fu *et al*. [4] presented a detailed discussion of three forms of the data reachability algorithm: *DataReach*, *M-DataReach* and *V-DataReach*, listed in increasing order of precision. One key contribution of work is V^a-*DataReach*, which 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 paths statically [24,25]. Bodik *et al*. [24] presented an algorithm for the C language to discover infeasible control-flow paths using branch correlation analysis. Our work is different in that we aim at eliminating infeasible inter-procedural paths due to polymorphism, an important object-oriented feature for Java. Souter and Pollock [25] presented an algorithm to detect type-infeasible call chains for Java. However, in their algorithm, only type information is propagated starting from each call site. Comparatively, our algorithm is more accurate because the fine-grained information (i.e., objects) is propagated using call-site specific points-to analysis.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have explored approaches for constructing an accurate *application call graph* for Java. We designed a new 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 from whole-program call graph. We also described how to automatically calculate a library call abstraction that can be applied in program slicing and dataflow testing, which we will explore as future work.

REFERENCES

1. Grove D, Chambers C. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*) 2001; **23**(6):685–746.
2. Ryder BG. Dimensions of precision in reference analysis of object-oriented programming languages. *Proceedings Twelfth International Conference on Compiler Construction*, invited paper, April 2003; 126–137.
3. Hind M. Pointer analysis: Haven't we solved this problem yet? *Proceedings ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001; 54–61.
4. Fu C, Milanova A, Ryder BG, Wonnacott D. Robustness testing of Java server applications. *IEEE Transactions on Software Engineering* 2005; **31**(4):292–311.
5. Zhang W, Ryder BG. Constructing accurate application call graphs for Java to model library callbacks. *Proceedings Sixth IEEE International Workshop on Source Code Analysis and Manipulation* (*SCAM 2006*), September 2006; 63–74.
6. Sharir M, Pnueli A. Two approaches to interprocedural data flow analysis. *Program Flow Analysis*: *Theory and Applications*. Prentice-Hall: Englewood Cliffs NJ, 1981; 189–234.
7. Shivers O. Control-flow analysis of higher-order languages. *PhD Thesis*, Carnegie Mellon University, 1991.
8. Scott ML. *Programming Language Pragmatics*. Morgan Kaufmann: Los Altos CA, 2000.
9. Choi JD, Gupta M, Serrano MJ, Sreedhar VC, Midkiff SP. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems* 2003; **25**(4):876–910.
10. Tip F, Palsberg J. Scalable propagation-based call graph construction algorithms. *Proceedings Conference on Object-oriented Programming*, *Languages*, *Systems and Applications*, October 2000; 281–293.
11. Rountev A, Milanova A, Ryder BG. Points-to analysis for Java using annotated constraints. *Proceedings Conference on Object-oriented Programming*, *Languages*, *Systems and Applications*, 2001; 43–55.
12. Lhoták O, Hendren L. Scaling Java points-to analysis using Spark. *International Conference on Compiler Construction*, 2003.
13. Specbench.org. Java client/server benchmarks. http://www.specbench.org/ [15 April 2006].
14. Zhang W, Ryder BG. A semantics-based definition for interclass test dependence. *DCS-TR-597*, Department of Computer Science, Rutgers University, January, 2006.
15. Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/ [15 January 2006].
16. bddbddb:BDD-Based Deductive DataBase. http://bddbddb.sourceforge.net/ [15 March 2006].
17. Milanova A, Rountev A, Ryder BG. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology* 2005; **14**(1):1–41.
18. Lhoták O, Hendren L. Context-sensitive points-to analysis: Is it worth it? *International Conference on Compiler Construction*, 2006.

19. Rountev A, Kagan S, Gibas M. Static and dynamic analysis of call chains in Java. *Proceedings ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004; 1–11.
20. Weiser M. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. *PhD Thesis*, University of Michigan, Ann Arbor, 1979.
21. Horwitz S, Reps T, Binkley D. Interprocedual slicing using dependence graph. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*) 1990; **12**(1):26–61.
22. Ranganath VP, Hatcliff JK. Slicing concurrent Java programs using Indus and Kaveri. *International Journal on Software Tools for Technology Transfer*, accepted. http://www.cis.ksu.edu/∼rvprasad/publications/stt05-submission.pdf [15 October 2006].
23. Rapps S, Weyuker E. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering* 1985; **SE-11**(4):367–375.
24. Bodik R, Gupta R, Soffa ML. Refining data flow information using infeasible paths. *Proceedings Sixth European Software Engineering Conference*. Springer: Berlin, 1997; 361–377.
25. Souter AL, Pollock LL. Characterization and automatic identification of type infeasible call chains. *Information and Software Technology* 2002; **4**(13):721–732.

AUTHORS' BIOGRAPHIES

Weilei Zhang is currently a PhD candidate in computer science at Rutgers University, New Brunswick, NJ. He received his BE degree in computer communication and MS degree in computer science from Beijing University of Posts and Telecommunications. His research interests include program analysis and its application in software engineering and compiler optimization.

Dr Barbara G. Ryder is a Professor of computer science at Rutgers University. She received her PhD degree in computer science at Rutgers in 1982. She worked in the 1970s at AT&T Bell Laboratories in Murray Hill, NJ. Dr Ryder's research interests include static and dynamic program analyses for object-oriented systems, focusing on usage in practical software tools.

Dr Ryder became a Fellow of the ACM in 1998, was selected as a CRA-W Distinguished Professor in 2004. and received the ACM SIGPLAN Distinguished Service Award in 2001. She was voted Professor of the Year for Excellence in Teaching by the Rutgers Computer Science Graduate Student Society in 2003, received a Leader in Diversity Award at Rutgers in 2006, and a Graduate Teaching Award from Rutgers Graduate School in 2007. She has been an active leader in ACM (ACM Council Member 2000–2008; Chair, FCRC 2003; Chair, ACM SIGPLAN 1995–1997) and has served as a Member of the Board of Directors of the Computer Research Association (1998–2001). Dr Ryder is an editorial board member of ACM Transactions on Programming Languages and Systems, IEEE Transactions on Software Engineering and Software, Practice and Experience. She has served on many program and conference committees, especially those sponsored by ACM SIGPLAN and ACM SIGSOFT and has been a panelist in the CRA Workshops on Academic Careers for Women and the New Software Engineering Faculty Symposia at ICSE.