A Generic Worklist Algorithm for Graph Reachability Problems in Program Analysis

Derek Rayside and Kostas Kontogiannis
Electrical & Computer Engineering
University of Waterloo
{drayside, kostas}@swen.uwaterloo.ca

Abstract

Many program analyses involve, or can be expressed in terms of, a graph reachability problem. We present a generic worklist-style algorithm capable of expressing and solving the graph reachability components of many such analyses. We compare our framework with the language reachability framework proposed by Thomas Reps, and show that some problems are expressible in both frameworks, and some problems can be expressed in only one of the frameworks.

Our two main case studies are Choi et al's escape analysis and Bacon & Sweeney's rapid type analysis (RTA). The reachability problems in these analyses can be directly expressed in the framework of our algorithm, but not in the language reachability framework. We discuss why this is the case, but important future work remains to formally characterize the kinds of problems that are amenable to our approach.

The paper also summarizes our experimental work with an implementation of RTA based on our generic algorithm.

1 Introduction

Program analysis is complicated. One of the ways in which researchers attempt to discover simplicity in this complexity is by examining the similarity and differences between analyses, and then developing a framework in which these can be expressed. One of the earliest, and best known, of these works in data-flow analysis is Gary Kildall's *Unified Approach to Global Program Optimization* from the first *POPL* conference [14]. More recently, Thomas Reps has shown how graph reachability problems in program analysis are related to formal languages, and how this relation leads to a more succinct expression and better understanding of certain program analyses [27].

In this paper we propose a generic worklist-style algorithm for graph reachability problems that can be specialized to solve specific problems, and to do so as efficiently as a hand-coded worklist algorithm. Specifically, we show how our algorithm can be used to express and compute the reachability problems in Choi et al's *escape analysis* [5] and Bacon & Sweeney's *rapid type analysis* (RTA) [1, 2]. Elsewhere [21], we have shown how our framework is also capable of expressing the reachability problems in the classic 'gen/kill bit-vector' intra-procedural data-flow analyses, program slicing [12], and program chopping [13] (a generalization of slicing).

In comparison to the formal language framework proposed by Reps, we show that certain problems are expressible and computable in both frameworks, and that some problems are expressible in only one of the frameworks. Our two case studies cannot be directly expressed in the language reachability framework, and our framework cannot express the problems that Reps is most interested in. We need to do further work to formally characterize the kinds of problems that can be expressed in our framework.

Organization of the Paper: Section two introduces algorithm A, our generic worklist algorithm, and its associated framework for expressing graph reachability problems. Section three summarizes the language reachability framework proposed by Reps [27]. Sections four and five explain Choi et al's escape analysis and Bacon & Sweeney's rapid type analysis, respectively, and discuss expressing them in both the algorithm A framework and the language reachability framework. Section six summarizes various experimental studies of RTA, including our own using algorithm A, and section seven concludes.

2 A Generic Worklist Algorithm

This section introduces algorithm A, our generic worklist algorithm (Figure 1), and the associated framework for expressing specific reachability problems. Like most

Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR'02)
1534-5351/02 $17.00 © 2002 IEEE

worklist-style algorithms, it involves two loops and a conditional: the outer loop continues while there is work to do; the inner loop iterates over the edges attached to the current working node; and the conditional ensures that the value associated with each node changes monotonically (which is necessary to ensure that the algorithm terminates).

Figure 1 Algorithm A: a generic worklist algorithm

```
Values ← InitialValues(G)
Worklist ← RootSet(G)
While hasMoreNodes(Worklist) Do
    n_i ← nextNode(Worklist)
    While hasMoreEdges(n_i) Do
        e ← nextEdge(n_i)
        τ ← type(e)
        n_j ← farNode(e, n_i)
        v'_j ← F(τ, v_i, v_j)
        If monotonicChange(v'_j, v_j) Then
            v_j ← v'_j
            addToWorklist(Worklist, n_j)
Return V
```

The major difference between algorithm A and many specialized worklist algorithms is captured in the line:

$$v'_j \leftarrow F(\tau, v_i, v_j)$$

F is the function that characterizes the specific analysis that the generic algorithm is currently performing; τ represents the type of the edge currently being traversed; v_i represents the value associated with the source node n_i; v_j represents the value associated with the target node n_j; v'_j is the value computed by F.

Customization Specific analyses can be implemented using this generic algorithm by providing definitions for F and $monotonicChange(v', v)$. This paper gives these definitions for Choi et al's escape analysis and RTA. Definitions for program slicing, program chopping (a generalization of slicing [13]), and the classic 'gen/kill bit-vector' intra-procedural data-flow problems are given in [21].

The algorithm can be used for analyses that traverse the graph forwards or backwards, or in both directions. For example, RTA is a forwards analysis, program slicing is a backwards analysis, and program chopping is a bidirectional analysis. The edge type τ is used to indicate edge direction: eg 'forward-call' and 'reverse-call' would be different types. The graph construction phase is responsible for typing the edges appropriately.

Termination Algorithm A will terminate if three conditions hold: 1. the values associated with each node are from a partially ordered domain (ie, there is a lattice that describes the domain); 2. all of the chains in the lattice between $\top$ and $\bot$ are finite; and 3. the values associated with each node change monotonically. Note that the function F does not have to return values that change monotonically, since this property is enforced by the conditional.

Computational Complexity The computational complexity of algorithm A is $O(e \times c)$, where e is the number of edges in the graph and c is length of the longest chain in the lattice that the values are taken from. Since c is a property of the specific analysis and is independent of the particular graph being analyzed, we may say that the algorithm is linear in the number of edges in the graph.

This analysis of the computational complexity assumes that all of the functions used in the algorithm execute in constant time.

Practical Performance Qadah et al's [20] empirical results show that the performance of algorithms such as this one are typically sub-linear in the number of edges because usually only a subset of the graph is reached, and so only some of the edges are traversed.

As is commonly known in data-flow analysis (eg [17]), the practical performance of worklist-style algorithms depends heavily on the worklist management strategy. We have developed a 'best-first' worklist management strategy for algorithm A, although our prototype implementation uses a simple breadth-first approach [21].

3 Program Analysis via Graph Reachability

This section summarizes the framework from Reps' excellent paper of the same title [27].

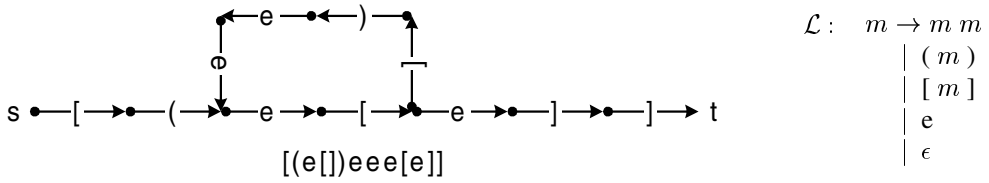
A path in a graph can be described with a word in a formal language. *Path expressions* are a well known example of this manner of description. In [27] a convention for labelling each edge in a graph with a character, and forming words by concatenating the characters of the edges along a path is adopted. Specifically, the following definition is used [27, paraphrased]:

$\mathcal{L}$-path: A path in G is an $\mathcal{L}$-path if its word w is a member of a language $\mathcal{L}$. (Where $\mathcal{L}$ is a formal language over alphabet Σ, and G is a graph whose edges are labelled with members of Σ.)

This definition is used to define various kinds of graph reachability problems:

All-pairs $\mathcal{L}$-path Problem: determine all pairs of nodes n_1 and n_2 such that there exists an $\mathcal{L}$-path in G from n_1 to n_2.

Figure 2 Example of a Context Free Language Reachability Problem from Reps [27]



Single-source/single-target $\mathcal{L}$-path Problem: determine whether there exists an $\mathcal{L}$-path in G from a given source node n_1 to a given target node n_2.

Single-source/multiple-target $\mathcal{L}$-path Problem: determine all nodes n_2 such that there exists an $\mathcal{L}$-path in G from a given source node n_1 to n_2.

Single-target/multiple-source $\mathcal{L}$-path Problem: determine all nodes n_1 such that there exists an $\mathcal{L}$-path in G from node n_1 to a given target node n_2.

Multiple-source/multiple-target $\mathcal{L}$-path Problem: determine all nodes n_2 such that there exists an $\mathcal{L}$-path in G from a source node n_1 to n_2, where $n_1 \in SourceSet$ and $SourceSet$ is given.

Multiple-target/multiple-source $\mathcal{L}$-path Problem: determine all nodes n_1 such that there exists an $\mathcal{L}$-path in G from n_1 to a target node n_2, where $n_2 \in TargetSet$ and $TargetSet$ is given.

Language Recognition Problem: Given a string w and a language $\mathcal{L}$, the problem is to determine if $w \in \mathcal{L}$. To treat this as a reachability problem, create a linear graph $s \rightarrow ... \rightarrow t$ that has $\mid w \mid$ edges, and label the i^{th} edge of with the i^{th} letter of w. There is an $\mathcal{L}$-path from s to t if and only if $w \in \mathcal{L}$. This is a special case of the single-source/single-target $\mathcal{L}$-path problem.

3.1 Classifying Program Analyses

Given the above definitions of various reachability problems, Reps divides program analyses according to the class of grammar required to describe them.

Regular Grammars The simplest *regular language reachability* problem is 'ordinary graph reachability', where the alphabet is $\Sigma = \{e\}$ and the language is $\mathcal{L} = e^*$. More interesting reachability problems may be described with richer regular grammars. For example, the two-pass reachability problem presented as a part of program slicing in Horwitz, Reps & Binkley [12] may be considered as a regular language reachability problem: label edges to be traversed on the first pass only with a; label edges to be traversed on the second pass only with b; label edges to be traversed on either pass with c; the language is $\mathcal{L} = (a|c)^*(b|c)^*$. This reachability problem is described with an imperative algorithm in [12]; our formulation is detailed in [21].

Algorithm A can express and compute reachability problems described by regular grammars.

Context Free Grammars Reps is particularly focused on program analyses that can be described with *context-free languages* because such languages allow one to describe call/return sequences in a program (the stack of the language recognizer traces potential call stacks in the program). A simple example adapted from [27] is represented here in Figure 2. A labelled graph is given on the left, and the grammar for the language $\mathcal{L}$ is given on the right (the non-terminal m stands for 'matched'). $\mathcal{L}$ may be described as the language that consists of strings of matched parentheses and square brackets, with zero or more e's interspersed. The word '$[(e[])eee[e]]$' indicates the only $\mathcal{L}$-path from s to t, which goes around the loop exactly once.

Reps is primarily concerned with inter-procedural analyses, including inter-procedural program slicing, certain data-flow analysis problems, one kind of heap shape analysis, and one kind of flow-insensitive points-to analysis. He shows how all of these can be described as *context-free language reachability problems*. The data-flow analyses amenable to this approach are characterized as IFDS: *inter-procedural, finite, distributive, subset* [27]. 'Finite' means that the set of data-flow facts is finite. 'Distributive' means that the data-flow functions distribute over the confluence operator, which is either set union or set intersection (in other words, the problems are 'separable'). The IFDS problems include the classic 'gen/kill bit-vector' problems, as well as many other non-gen/kill problems, such as possibly-uninitialized variables, truly-live variables, and copy-constant propagation [27].

As is well known, context-free languages require $O(n^3)$ time to recognize in general [32], and Reps argues that this provides important theoretical insight into why certain program analyses require $O(n^3)$ algorithms. Context-free languages can be recognized in sub-cubic time if the graph is acyclic [31], but this is rarely the case in program analysis.

Algorithm A can neither express nor compute reachability problems described by context-free grammars.

4 Escape Analysis

The purpose of *Escape Analysis* is to determine if an object is contained within a single method or a single thread. If an object is contained within a single method then it may be automatically stack allocated instead of heap allocated, and if it is contained within a single thread then all of its synchronization operations may be removed. Escape analysis is somewhat similar to points-to or live-variable analysis.

Escape analysis for Java is a topic that has attracted quite a bit of attention recently (eg, there are four papers in [18]). This section explains the reachability problem in the escape analysis proposed by Choi et al from IBM Research [5], which is based on a program abstraction called a *Connection Graph*. This escape analysis proposal describes two main phases: first, the connection graph is constructed, and then it is traversed to determine which objects escape. The mechanics of graph construction is described fully in [5], and is not our main concern here. The remainder of this section represents the connection graph definition and describes the reachability problem.

Connection Graph The connection graph is defined as:

$$CG = (N_o \cup N_r \cup N_f \cup N_g, E_r \cup E_d \cup E_f)$$

Where N_o is the set of nodes that represent objects, N_r is the set of nodes that represent reference variables, N_f is the set of nodes that represent instance fields, and N_g is the set of fields that represent static fields (ie, global variables); E_r represents the set of 'points-to' edges, E_d represents the set 'deferred' edges, and E_f is the set of 'field' edges.

Three other sets of nodes, other than those used in the definition, are identified for initialization purposes: $N_{runnable}$ represents all objects that instantiate a class that implements the `Runnable` interface (ie, all thread objects); $N_{finalizable}$ represents all objects that instantiate a class with a non-trivial finalizer/destructor method; $N_{paramater}$ represents all objects that are passed between methods as parameters.

4.1 Reachability Problem

The reachability problem to be solved on the connection graph is defined with respect to a three element lattice [5]: $NoEscape$ ($\top$), $ArgEscape$, and $GlobalEscape$ ($\bot$). $NoEscape$ means the object does not escape the method that allocates it; $ArgEscape$ means the object may escape the method that allocates it, but does not escape the thread that allocates it; $GlobalEscape$ means that the object may escape both the method and the thread that allocate it. The reachability problem is intuitively described like so:

> *Let CG be a connection graph for method M, and let O be an object node in CG. If O can be reached in CG from any node whose escape state is not $NoEscape$, then O escapes M.* The intuition easily extends to the escapement of an object from a thread. [5, emphasis in original]

Graph Initialization Table 1 lists the initial connection graph node value assignments used for escape analysis.

Table 1 Connection graph initialization values [5]

Nodes	*Initial Value*
N_o	$NoEscape$
N_r	$NoEscape$
N_f	$NoEscape$
N_g	$GlobalEscape$
$N_{runnable}$	$GlobalEscape$
$N_{finalizable}$	$GlobalEscape$
$N_{paramater}$	$ArgEscape$

The Original Algorithm Choi et al's original worklist-style algorithm is represented in Figure 3. This algorithm involves two applications of a simple worklist algorithm: the first pass propagates the $GlobalEscape$ values, and the second pass propagates the $ArgEscape$ values. This division of labour saves some computational effort: if an object may escape its allocating thread (ie, $GlobalEscape$), then by definition it may escape its allocating method (ie, $ArgEscape$). So, by propagating the $GlobalEscape$ values first the algorithm eliminates redundancy from the reachability computation.

Figure 3 Escape Analysis Reachability [5]

```
// First Pass: propagate GlobalEscape values
Worklist ← GlobalEscapeNodes(G)
While hasMoreNodes(Worklist) Do
    m ← nextNode(Worklist)
    ForEach outgoing edge m → n Do
        If (State(n) ≠ GlobalEscape) Then
            State(n) ← GlobalEscape
            addToWorklist(Worklist, n)

// Second Pass: propagate ArgEscape values
Worklist ← ArgEscapeNodes(G)
While hasMoreNodes(Worklist) Do
    m ← nextNode(Worklist)
    ForEach outgoing edge m → n Do
        If (State(n) > ArgEscape) Then
            State(n) ← ArgEscape
            addToWorklist(Worklist, n)
End
```

An Equivalent Algorithm A functionally equivalent single pass algorithm is presented in Figure 4. There are three changes of note: 1. the worklist is initialized with all nodes that have state not equal to $NoEscape$; 2. the conditional test compares the states of both the source and target nodes, rather than the state of the target node and a constant; 3. the state of the target node is set to the state of source node, instead of to a constant. Replacing the constants with values from the source node makes this algorithm more general: it is now independent of the lattice, whereas the lattice is hard-coded into the original algorithm. Algorithm A is essentially a further generalization of the one in Figure 4.

This functionally equivalent algorithm is not computationally equivalent unless a best-first worklist management strategy is used. We have described such a strategy for algorithm A in [21]. The best-first strategy ensures that the $GlobalEscape$ values are propagated first, and so avoids redundancy in the same manner as the original escape analysis reachability algorithm.

Figure 4 Equivalent Reachability Algorithm

```
Worklist ← GlobalOrArgEscapeNodes(G)
While hasMoreNodes(Worklist) Do
    m ← nextNode(Worklist)
    ForEach outgoing edge m → n Do
        If (State(n) > State(m)) Then
            State(n) ← State(m)
            addToWorklist(Worklist, n)
End
```

Expression with Algorithm A Since the lattice is totally ordered, each position can be represented with an integer: $GlobalEscape = 0$, $ArgEscape = 1$, and $NoEscape = 2$. The $monotonicChange(v', v)$ function can then be defined as the less-than ($<$) relation.

The function F can be defined simply as the identity function: that is, F always returns the v_i parameter (the value associated with the source node). The other two parameters, edge type τ and target node value v_j, are disregarded. This definition simply propagates the value from the source node to the target node.

Expression with Language Reachability Each pass in the two-pass escape analysis reachability algorithm (Figure 3) is (almost) equivalent to a *multiple-source* ordinary reachability problem (ie, $\mathcal{L} = e^*$). When considered as a language reachability problem, the set of $ArgEscape$ target nodes must be subtracted from the set of $GlobalEscape$ target nodes, since $GlobalEscape$ supersedes $ArgEscape$. The algorithm in Figure 3 avoids having to compute this set subtraction because of the conditional in the second pass.

So, the language reachability approach is not capable of expressing this escape analysis reachability problem with a single application of a single language: the same language ($\mathcal{L} = e^*$) must be applied to two different source sets, and then one result set must be subtracted from the other. There is no way to express either the repeated application or the set subtraction. We think that the root of the difficulty is that the domain of potential node values has three members, and this is incompatible with viewing the reachability problem as a language recognition problem: a word is either a member of a language or it isn't, and that means a domain with only two members.

5 Rapid Type Analysis

Rapid Type Analysis (RTA) is a fast and effective means for statically resolving polymorphic method invocations in class-based, statically typed, object-oriented programming languages with method overriding semantics [1, 2]. RTA is a flow- and context-insensitive whole program analysis.

The basic idea of RTA is to prune the results of *Class Hierarchy Analysis* (CHA) [7, 8] with information about instantiation. This basic idea can be explained by analyzing the sample program listed in Figure 5. The program contains one polymorphic invocation, `s.draw()`, in the `Main::foo(Shape)` method. The purpose of call graph construction techniques such as CHA and RTA is to determine the set of potential targets for such invocations.

A class hierarchy analysis of the example program reveals that the signature for the `draw` method is contained in the abstract class `Drawable`, which is the immediate super-class of `Shape`. `Shape` has four sub-classes (`Circle`, `Triangle`, `Rectangle`, and `Square`), three of which implement the `draw` method. CHA identifies these three implementations of `draw` as possible targets for the invocation `s.draw()`.

RTA tries to eliminate some of the CHA-identified possible targets by using information about instantiation. Starting at the program entry point(s), in this case `main`, it can be seen that the example program only instantiates `Circle` and `Square`. Therefore `Triangle::draw` cannot be executed, since neither `Triangle` nor any of its subclasses is ever instantiated. `Rectangle::draw` may be executed because its sub-class, `Square`, is instantiated and does not re-implement `draw`.

So, a method implementation may be executed if it is identified as a potential invocation target by CHA and it is associated with an instantiated class (ie, it is the method that will be executed for objects of that class). In reachability terms, a method implementation may be executed if it can be reached from the program entry point(s) via both an 'invocation' path and an 'instantiation' path. Abstractly, the reachability problem is a conjunction of paths.

Figure 5 Example program for explaining Rapid Type Analysis

```
package ca.uwaterloo.swen;
import ca.uwaterloo.graphics;

public class Main {
  public static void main(String[] args) {
    Shape shape;
    shape = new Circle();
    foo(shape);
    shape = new Square();
    foo(shape);
  }
  static void foo(Shape s) {
    s.draw();
  }
}
```

```
package ca.uwaterloo.graphics;

abstract class Drawable {
  abstract void draw();
}
abstract class Shape extends Drawable {}
class Circle extends Shape {
  void draw() { printf("Circle"); }
}
class Triangle extends Shape {
  void draw() { printf("Triangle"); }
}
class Rectangle extends Shape {
  void draw() { printf("Rectangle"); }
}
class Square extends Rectangle {}
```

5.1 Expressing RTA with Algorithm A

Our implementation of RTA with algorithm A is based on a dependency graph model for Java bytecode that we have described in detail in [21, 23], and partially in [22, 24]. Basically, nodes represent types, fields, and methods; edges represent declaration, extension, invocation, instantiation, type tests, fetch, store, etc. The complete model has 41 different types of edges, including six kinds of edges to represent method invocation. These static dependency graphs are built from the Java bytecode, and the construction incorporates CHA, as discussed in [23, 21].

Lattice The original formulation of RTA in [1] includes a worklist-style algorithm, but not an explicit lattice.

RTA can also be thought of as an analysis to determine which parts of the program may be exercised during execution, and it is often used for this purpose (eg, [29, 23]). The lattice in Table 2 is designed from this perspective, and the $monotonicChange(v', v)$ predicate can be defined as the greater-than ($>$) relation for use with these values.

Table 2 Lattice for RTA

Name	*Description*	*Value*
new	directly instantiated	5
reqd	required for execution	4
invoked	may be the target of a (possibly) live polymorphic invocation	3
member	may be required by an instantiated class	2
file	in same file as a required class	1
$\perp$	not required	0

The **invoked** lattice value indicates that the program element (ie, method) may by the target of a polymorphic method invocation. The **member** lattice value indicates that the program element (ie, method) may be required by an instantiated class. If a method implementation is both **invoked** and a **member**, then it is **reqd**: this is the conjunction of paths that characterizes RTA.

Function F The function F for RTA can be broken down into six cases based on the type of the edges. We denote these six cases as F^0, $F^{default}$, F^{file}, F^{new}, $F^{member}$and F^{invoke}. The first four are given in Table 3, F^{member} is given in Table 4, and F^{invoke} is given in Table 5. The first four functions depend only on the source node value; the value currently associated with the target node is disregarded. The last two depend on both the current source node and target node values. All of these functions can be (and have been) implemented as constant time array lookups.

The 'not required' lattice value is denoted with '–' in Tables 3, 4 & 5. Recall that the F functions do not have to return monotonically changing values: this property is enforced by the $monotonicChange(v', v)$ predicate. Therefore, if an F function returns a value that causes the $monotonicChange(v', v)$ predicate to fail, it simply means that the source node is transmitting no new information to the target node. This is what the '–' return values in Tables 3, 4 & 5 indicate.

Table 3 RTA F^0, $F^{default}$, F^{file}, and F^{new}

Source	$\mapsto$	F^0	$F^{default}$	F^{file}	F^{new}
–	$\mapsto$	–	–	–	–
file	$\mapsto$	–	–	–	–
member	$\mapsto$	–	–	–	–
invoke	$\mapsto$	–	–	–	–
reqd	$\mapsto$	–	reqd	file	new
new	$\mapsto$	–	reqd	file	new

Table 4 RTA Function F^{member}

Source	*Target*	$\mapsto$	F^{member}
–	*	$\mapsto$	–
file	*	$\mapsto$	–
member	*	$\mapsto$	–
invoke	*	$\mapsto$	–
reqd	*	$\mapsto$	file
new	–	$\mapsto$	member
new	file	$\mapsto$	member
new	member	$\mapsto$	member
new	invoke	$\mapsto$	reqd
new	reqd	$\mapsto$	reqd
new	new	$\mapsto$	reqd

Table 5 RTA Function F^{invoke}

Source	*Target*	$\mapsto$	F^{invoke}
–	*	$\mapsto$	–
file	*	$\mapsto$	–
member	*	$\mapsto$	–
invoke	*	$\mapsto$	–
reqd	–	$\mapsto$	invoke
reqd	file	$\mapsto$	invoke
reqd	member	$\mapsto$	reqd
reqd	invoke	$\mapsto$	invoke
reqd	reqd	$\mapsto$	reqd
reqd	new	$\mapsto$	reqd
new	*	$\mapsto$	–

The functions F^{member} and F^{invoke} are moderately more complicated than the other four functions. However, they are still fairly simple: they are both roughly equivalent to the truth table for logical conjunction, adjusted for the fact that we are dealing with a domain of six values, rather than simply the two values zero and one.

A table associating these six functions with the 41 edge types in the graph schema is presented in [21]. In summary, F^{file} is associated with field and constructor declarations; F^{new} is associated with instantiations; F^{member} is associated with method declarations; and F^{invoke} is associated with polymorphic invocations. $F^{default}$ is associated with most of the other arc types. The exact details of the associations depend on our class hierarchy analysis implementation. The interested reader is referred to [21] for details.

5.2 Expressing RTA with Language Reachability

RTA can be expressed as a language reachability problem, using a dependency graph model similar to the one used above. To be expressed in this way, RTA requires the combination of two regular languages, named $\mathcal{L}_{invoke}$ and $\mathcal{L}_{new}$ here in Figure 6.

Figure 6 Languages Expressing RTA Reachability Problem

$$\mathcal{L}_{invoke} = invoke^*$$
$$\mathcal{L}_{new} = invoke^*\ new\ extends^*\ declares$$

In terms of graph reachability, RTA represents a conjunction of paths: a method may be executed *if* it is reachable via a chain of invocations *and* some class that declares or 'inherits' it is instantiated. The first part of this conjunction is represented by $\mathcal{L}_{invoke}$, and the second part is represented by $\mathcal{L}_{new}$. Each of these languages should be applied as a multiple-source reachability problem.

From the previous example program, it can be seen that `Rectangle::draw` may be executed because it can be reached in both of these ways. The specific paths are shown in Figure 7 by w_{invoke} and w_{new}, where $w_{invoke} \in \mathcal{L}_{invoke}$ and $w_{new} \in \mathcal{L}_{new}$. Each arc is indicated by an arrow with a character over it indicating the type of the arc: i means 'invoke', n means 'new', e means 'extends', and d means 'declares'.

Figure 7 Words reaching `Rectangle::draw`

$$w_{invoke} = \text{main} \xrightarrow{i} \text{foo} \xrightarrow{i} \text{Rectangle} :: \text{draw}$$
$$w_{new} = \text{main} \xrightarrow{n} \text{Square} \xrightarrow{e} \text{Rectangle} \xrightarrow{d} \text{Rectangle} :: \text{draw}$$

Once it is determined that `Rectangle::draw` may be executed, then it must be considered as a starting point for $\mathcal{L}_{invoke}$ and $\mathcal{L}_{new}$ (as `main` currently is). This requirement indicates that not only must these two languages be combined, but this combination must be repeatedly applied until no more potentially executing methods are discovered. This successive re-application makes this problem different than a regular multiple-source reachability problem, because the set of source nodes grows during the analysis (eg `Rectangle::draw` becomes a source node). For these two reasons RTA cannot be expressed solely in terms of language reachability: extra work is necessary for the conjunction of paths and the growing source set.

6 Summary of Experiments

This section summarizes our experimental results from [21], in context with studies from (in chronological order): Bacon & Sweeney [1, 2], Porat et al [19], Tip et al [29], Sundaresan et al [28], and Tip & Palsberg [30]. This section is not intended as a critical comparison of these studies but, rather, to demonstrate that our implementation produces results similar to those obtained in other studies.

Table 6 Summary of studies of RTA's effectiveness for static resolution of polymorphic method calls

Study	*Language*	*Lib. Incl.*	*min*	*max*	*avg*
Bacon & Sweeney [1, 2]	C++	×	28%	100%	75%
Porat et al [19]	Java	√	53%	99%	99%
Sundaresan et al [28]	Java	√	8%	81%	40%
Tip & Palsberg [30]	Java	×	81%	97%	92%
Rayside & Kontogiannis [21]	Java	√	83%	89%	85%

Table 7 Summary of studies of RTA's effectiveness at application/library-subset extraction

Study	*Language*	*Lib. Incl.*	*Measure*	*min*	*max*	*avg*
Bacon & Sweeney [1, 2]	C++	×	bytes	1%	63%	25%
Bacon & Sweeney [1, 2]	C++	×	methods	6%	89%	40%
Porat et al [19]	Java	×	methods	10%	65%	43%
Tip et al [29]	Java	×	methods	10%	89%	41%
Tip et al [29]	Java	×	bytes	13%	90%	53%
Rayside & Kontogiannis [21]	Java	×	bytes	1%	76%	28%
Rayside & Kontogiannis [21]	Java	√	bytes	65%	97%	80%

Our contribution is the novel way in which we have implemented RTA using algorithm A, not RTA itself.

The two most common objectives of RTA are static resolution of polymorphic method invocations and application/library-subset extraction. (We have also examined RTA for the purposes of automatic clustering [26] and program understanding [25].)

All of the measurements presented here are with respect to the statically available code, not against what a dynamic trace considers to be 'live'. Bacon & Sweeney [1, 2] report in both ways.

Call-Graph Construction RTA is fairly effective at resolving virtual method calls to a single target, as is shown in Table 6. The average result of the five studies summarized in Table 6 is that RTA usually resolves about 80% of the virtual method calls to a single target. It is important to note that this average is indicative only and not critical: each of these studies measures slightly different things on different benchmarks. Some of the peculiarities in measurement include: whether the libraries the program depends on are included (this is indicated by the *lib. incl.* column in Table 6); whether abstract methods are counted ([30] excludes them); whether final methods are counted ([19] excludes them); and whether the analysis is based on source code, byte code, or some other intermediate format (eg, [28] is based on the JIMPLE intermediate format, which incorporates some type inference of the stack [9]).

One of the peculiarities of our study is that we group call sites with identical target signatures by introducing a *polymorphic choice vertex*, in a manner similar to [15]. This technique reduces the overall size of our graph and the time taken for our class hierarchy analysis. Table 6 reports the number of call site groups resolved to a single target in our study, rather than the number of actual call sites.

The studies from Sundaresan et al [28] and Tip & Palsberg [30] propose algorithms that are more effective than RTA and have similar cost — VTA and XTA, respectively.

Application Extraction Since RTA builds a call graph by identifying which methods may be executed (in a conservative fashion), it also identifies which methods may never be executed by a particular program. Table 7 summarizes the results of four studies. As with Table 6, this comparison is indicative only and not critical.

There are two main measurements that can be used to assess the volume of code removed: *bytes* of object code or number of *methods*. Some studies report only one of the measurements, other studies report both, as is indicated in Table 7. On average, 41% of the methods were removed from the application, which constituted 35% of the application size (these are averages of the results that do not include the libraries the application depends on).

In [21, 24, 23] we show that applications typically use less than half of the code (in bytes) of the libraries that they depend on. For example, the maximum 97% reported in Table 7 is for HelloWorld, which uses only 3% of the standard Java library; our bytecode analyzer uses only 7% of the standard library.

The study of Tip et al [29] reports results from their Jax

tool which also incorporates a number of more advanced analyses and transformation, such as *class hierarchy specialization*. Consequently, Jax eliminates more bytes of code than tools based solely on RTA.

7 Conclusion

We have proposed algorithm A, a generic worklist-style algorithm for graph reachability problems in program analysis, and its associated framework for expressing specific reachability problems. We have also summarized experimental work comparing our algorithm A based implementation of RTA with other specialized implementations of RTA.

We have compared our framework to the language reachability framework proposed by Reps [27], and shown that while some problems may be expressed in both frameworks, there are also problems that can be expressed in only one of the two frameworks. Both frameworks are capable of expressing reachability problems that can be described by a single regular language. Reps is primarily interested in modeling inter-procedural program analyses with context-free languages, and our framework does not support these problems. Our two case studies, Choi et al's *escape analysis* [5] and Bacon & Sweeney's *rapid type analysis* [1, 2], can be directly expressed in the framework of algorithm A but not in the language reachability framework.

The critical feature of the escape analysis reachability problem is that it the lattice of values that may be associated with a node has three members: $NoEscape$, $ArgEscape$, and $GlobalEscape$. Modeling a reachability problem as a language recognition problem limits one to a domain of two values: accept or reject.

The rapid type analysis reachability problem has two critical features: a conjunction of paths and a growing source set. The basic idea of RTA is that a method implementation may be executed if it is reached from the program entry point(s) via both an invocation path and an instantiation path. Once a method implementation has been reached by both paths, then it must be considered as a new origin point for other invocation and instantiation paths. The invocation paths and instantiation paths can be individually modeled in the language reachability framework, but the way in which they must be combined cannot.

Future Work The most important future work is to try and characterize the problems that can be expressed with algorithm A more formally. Two possible avenues of enquiry here are graph grammars and the Tarski relational algebra approach advocated by Ric Holt [10].

A number of people have mentioned to us that there may be some related work in model-checking, and we are looking into this.

Acknowledgments

We thank IBM's Centre for Advanced Studies and High Performance Java S/390 Compiler Group at the IBM Toronto Laboratory for their support of this work. Scott Kerr was instrumental in the early development of this project and has commented on the thesis this paper is based on [21]. The official readers of the thesis at the University of Waterloo, Professors Joanne Atlee and Rudolph Seviora, have provided important comments. Jens Krinke has also made a few insightful comments on an earlier version of this work. Finally, the anonymous referees also provided very helpful and useful comments.

References

[1] David F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California at Berkeley, December 1997. Supervised by Susan Graham. UCB/CSD-98-1017.

[2] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In Coplien [6], pages 324 – 341.

[3] Michael Blaha, Alex Quilici, and Chris Verhoef, editors. *Proceedings of the 5^{th} IEEE Working Conference on Reverse Engineering (WCRE)*, Honolulu, October 1998.

[4] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[5] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In Northrop [18], pages 1–19.

[6] James Coplien, editor. *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, San Jose, California, October 1996.

[7] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *ECOOP'95*, Århus, Denmark, August 1995. Springer-Verlag. LNCS 952.

[8] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In Coplien [6], pages 292–305.

[9] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *SAS'00*, pages 199–219, Santa Barbara, California, June 2000.

[10] Richard C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In Blaha et al. [3], pages 210–219.

[11] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *PLDI'88*, pages 35–46, Atlanta, Georgia, USA, June 1988. Conference version of [12].

[12] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1), January 1990. Journal version of [11].

[13] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. In *FSE'94*, pages 2–10, New Orleans, Louisiana, December 1994.

[14] Gary A. Kildall. A unified approach to global program optimization. In *POPL'73*, pages 194–206, Boston, MA, October 1973.

[15] Loren Larsen and Mary Jean-Harrold. Slicing object-oriented software. In *ICSE'96*, pages 495 – 505, March 1996.

[16] Doug Lea, editor. *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, Minneapolis, Minnesota, October 2000.

[17] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[18] Linda Northrop, editor. *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, Denver, Colorado, November 1999.

[19] Sara Porat, Bilha Mendelson, and Irina Shapira. Sharpening global static analysis to cope with Java. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON'98*, pages 303 – 316, Toronto, December 1998.

[20] Ghassan Z. Qadah, Lawrence J. Henschen, and Jung J. Kim. Efficient algorithms for the instantiated transitive closure queries. *IEEE Transactions on Software Engineering*, 17(3):296–309, March 1991. Reprinted in [4].

[21] Derek Rayside. A generic worklist algorithm for graph reachability problems in program analysis. Master's thesis, University of Waterloo, 2001. Supervised by Kostas Kontogiannis.

[22] Derek Rayside, Scott Kerr, and Kostas Kontogiannis. Change and adaptive maintenance detection in Java software systems. In Blaha et al. [3], pages 10–19.

[23] Derek Rayside and Kostas Kontogiannis. Extracting Java library subsets for deployment on embedded systems. *Science of Computer Programming*. To appear. An extended version of [24].

[24] Derek Rayside and Kostas Kontogiannis. Extracting Java library subsets for deployment on embedded systems. In Paolo Nesi and Chris Verhoef, editors, *CSMR'99*, pages 102–110, Amsterdam, March 1999. Best Paper Award.

[25] Derek Rayside and Kostas Kontogiannis. On the Syllogistic Structure of Object-Oriented Programming. In Hausi Müller, Mary-Jean Harrold, and Willhelm Shäfer, editors, *ICSE'01*, pages 113–122, Toronto, Canada, May 2001.

[26] Derek Rayside, Steve Reuss, Erik Hedges, and Kostas Kontogiannis. The effect of call graph construction algorithms for object-oriented programs on automatic clustering. In Margaret-Anne Storey, Anneliese von Mayrhauser, and Harald Gall, editors, *IWPC'00*, pages 191–200, Limerick, Ireland, June 2000.

[27] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11):701–726, November 1998.

[28] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, and Etienne Gagnon. Practical virtual method call resolution for Java. In Lea [16], pages 264 – 280.

[29] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. In Northrop [18], pages 292 – 305.

[30] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In Lea [16], pages 281 – 293.

[31] L.G. Valiant. General context-free recognition in less than cubic time. *J. Comp. Syst. Sci.*, 2(10):308–315, April 1975.

[32] D.H. Younger. Recognition and parsing of context-free languages in time n^3. *Inf. and Cont.*, 10:189–208, 1967.