Static Analysis for Dynamic Coupling Measures

Yin Liu Ana Milanova

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
{liuy,milanova}@cs.rpi.edu

Abstract

Coupling measures have important applications in software development and maintenance. They are used to reason about the structural complexity of software and have been shown to predict quality attributes such as fault-proneness, ripple effects of changes and changeability. Traditional object-oriented coupling measures do not account for polymorphic interactions, and thus underestimate the complexity of classes and fail to properly predict their quality attributes.

To address this problem Arisholm et al. [3] define a family of *dynamic coupling measures* that account for polymorphism. They collect dynamic coupling measures through dynamic analysis and show that these measures are better indicators of complexity and better predictors of quality attributes than traditional coupling measures.

This paper presents a new approach to the computation of dynamic coupling measures. Our approach uses *static analysis*, in particular *class analysis*, and is designed to work on incomplete programs. We perform experiments on several Java components and present a precision evaluation which shows that inexpensive class analysis such as RTA computes dynamic coupling measures with almost perfect precision. Our results indicate that inexpensive static analysis may be used as a more convenient, more practical and more precise alternative to dynamic analysis for the purposes of computation of dynamic coupling measures.

Copyright © 2006 Yin Liu and Ana Milanova. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

1 Introduction

Coupling measures are typically based on some form of static code analysis and their primary goal is to quantify the connectedness of a class to other classes. For example, class A is coupled to class B if some method in A calls a method in B. Coupling measures have important applications in software development and maintenance. They are used to help developers, testers and maintainers reason about software complexity and software quality attributes. Coupling measures have been studied extensively and have been used to assist maintainers in tasks such as impact analysis, assessing the fault-proneness of classes, fault prediction, ripple effects, and changeability among others [8, 9, 29, 30, 17]. Therefore, coupling measures assist developers, testers and maintainers in reasoning about the software and in predicting the needs for code inspection, testing and debugging.

1.1 Dynamic Coupling Measures

Traditional coupling measures [10, 11, 8] take into account only "static" couplings. They do not account for "dynamic" couplings due to polymorphism and may significantly underestimate the complexity of software and misjudge the need for code inspection, testing and debugging. As an illustrating example, consider the standard Bridge structure [15] shown in Figure 1. Traditional coupling measures [10, 11, 8] would count only *one* coupling due to polymorphic call `imp.DevM()` in method `A:m`, namely a coupling of `A` to `Imp`. In fact, the actual complexity of the call is considerably higher since the polymorphic call can couple "dynamically" each concrete `A` to each concrete `Imp` for a total of *six* couplings: ⟨`A1,Imp1`⟩, ⟨`A1,Imp2`⟩, ⟨`A1,Imp3`⟩, ⟨`A2,Imp1`⟩, ⟨`A2,Imp2`⟩ and ⟨`A2,Imp3`⟩.

Arisholm et al. [2, 4, 3] address this problem by proposing several sets of dynamic coupling measures specifically designed to account for polymorphic interactions. They use dynamic analysis to capture the dynamic couplings: they instrument the program, then execute it with all available inputs and collect dynamic measures over all executions. Arisholm et al. perform detailed statistical analysis that shows that these measures are better indicators of complexity and better predictors of quality attributes than traditional coupling measures.

1.2 Static Class Analysis

However, dynamic analysis has several disadvantages: (i) it is relatively slow, (ii) requires a complete program, and (iii) produces incomplete results as the couplings output by the dynamic analysis are valid for particular inputs and executions of the program.

The goal of our work is to utilize static analysis as an alternative to dynamic analysis for the computation of dynamic coupling measures. The static analyses considered in this paper may have several advantages over dynamic analysis: (i) they are practical, (ii) work on incomplete programs and (iii) produce results valid over all program executions (Section 2 additionally argues the suitability of static analysis).

We propose a static analysis framework for the computation of the dynamic coupling measures from [3] for strictly-typed languages such as Java. The framework is parameterized by *class analysis*, which determines the classes of the objects a reference variable or a reference object field may refer to; the framework can be instantiated with a wide variety of class analyses of varying degrees of precision and cost [1, 5, 16, 13, 18, 7, 19, 22, 25, 27, 28]. We use the class analysis solution to approximate the interactions of run-time objects and infer the possible dynamic couplings. Our approach works on incomplete programs. This is an important feature because it is essential to be able to perform separate analysis of software components. For example, it is typical to have to assess the quality attributes of a component without having access to the clients of that component.

We have instantiated the framework with Class Hierarchy Analysis (CHA) [12] and Rapid Type Analysis (RTA) [5], two well-known class analyses at the lower end of the cost/precision spectrum; we approximate dynamic couplings with the two instantiations. We present empirical results on several components. We present a precision evaluation which shows that one of our analyses, namely the one based on RTA, achieves almost perfect precision—that is, it computes almost exactly the same couplings that would have been identified dynamically with the most complete suite of test cases written on top of the component. These results indicate that dynamic coupling measures can be measured precisely with a simple and inexpensive static analysis. Thus, our analysis can be easily incorporated in software tools; it can assist developers, testers and maintainers in assessing software quality and in predicting quality attributes such as changeability and fault proneness.

This work has the following contributions:

- We propose a static analysis framework parameterized by *class analysis*, for the computation of dynamic coupling measures. Our analysis framework is designed to handle incomplete programs.
- We present an empirical study that evaluates two instantiations of the framework

on several Java components.

2 The Case for Static Analysis

Arisholm et al. [3] capture dynamic coupling measures through dynamic analysis; however, they observe that dynamic analysis has (arguably substantial) drawbacks. First, it is relatively slow because it requires multiple executions of the program with multiple inputs and each execution incurs the overhead of instrumentation. Second, the engineering task of building an instrumentation framework is relatively complex. Third, dynamic analysis always requires a complete program while in many cases it is necessary to use coupling measures to assess quality attributes of software components (i.e., incomplete programs). Fourth, the results that are obtained may be incomplete as they are based on particular runs with particular inputs. As an example, suppose that there are two clients of the Bridge structure in Figure 1, one that instantiates `A1` with `Imp1` and one that instantiates `A1` with `Imp2`. Running the two clients will count dynamic couplings ⟨`A1,Imp1`⟩ and ⟨`A1,Imp2`⟩ as due to call `imp.DevM()`. Thus, dynamic analysis will omit four valid couplings. One would need at least six clients, and six runs, in order to capture correctly all possible dynamic couplings due to this call.

Static analysis, and in particular class analysis, presents a viable alternative to dynamic analysis for the purposes of the computation of dynamic coupling measures. Using class analysis, one can approximate the possible classes of each reference variable and reference object field and reason about possible dynamic couplings. For the running example, a class analysis may determine that the possible run-time classes for `imp` are `Imp1`, `Imp2` and `Imp3` and the possible run-time classes for implicit parameter $\mathtt{this}_{\mathtt{A:m}}$ are `A1` and `A2`. This information correctly yields that call `tmp.DevM()` may trigger six dynamic couplings (clearly, $\mathtt{this}_{\mathtt{A:m}}$ refers to the receivers of `A:m` and it captures the classes of the caller; `imp` captures the classes of the callee). Static analysis has important advantages over dynamic analysis when computing dynamic coupling measures. First, the static analyses considered in this paper are relatively inexpensive and likely much more practical than dynamic analysis. Second, these analyses are easy to implement; it is likely that they are implemented for the purposes of essential tasks such as call graph construction, and the analysis for the computation of dynamic coupling measures will simply reuse available results. Third, the static analysis considered in this paper can be adapted to work on incomplete programs (i.e., software components), which is an important requirement. Fourth, the static analysis is conservative—that is, it is guaranteed that all possible run-time couplings will be reported.

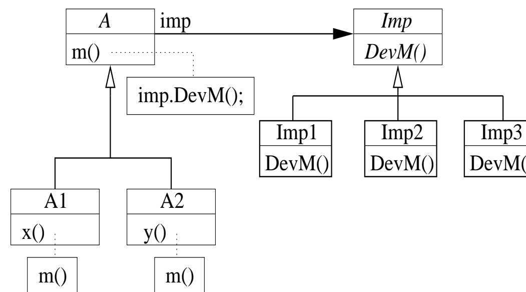

Figure 1: The Bridge pattern structure

However, static analysis can be overly conservative and report infeasible couplings—that is, couplings that cannot happen during any execution of the program. Clearly, overly conservative analysis may overestimate the possible couplings and software complexity and thus undermine the usefulness of the coupling measures. We believe that in order to use static analysis as a viable alternative to dynamic analysis, the static analysis must provide a precise approximation of the dynamic couplings.

3 Problem Statement

The goal of our work is to design static analysis that computes dynamic coupling measures precisely; the analysis must work on incomplete programs. This analysis problem fits into a general framework for analysis of incomplete

programs due to Nasko Rountev [23, 26, 24]. The input to the analysis contains a set *Cls* of interacting Java classes. A subset of *Cls* is designated as the set of *accessible classes*; these are classes that may be accessed by unknown client code from outside of *Cls*. Such client code can only access fields and methods from *Cls* that are declared in some accessible class; these accessible fields and methods are referred to as *boundary fields* and *boundary methods*.

3.1 Dynamic Coupling Measures

The dynamic coupling measures we study were defined and studied in detail by Arisholm et al. [2, 3]. Our work focuses on the *object coupling measures* which capture polymorphic interactions. The measure of highest granularity, denoted by *IC_OD* in [3], records a tuple $\langle C_1, m_1, I, C_2, m_2\rangle$ when the dynamic analysis encounters an execution of method m_1 with receiver of type C_1 *and* during this execution call site I in m_1 invokes method m_2 with receiver of class C_2. Recall the example in Figure 1 and assume that call site `imp.DevM()` in method `A:m` has number 1. Assuming that there are tests that ensure complete coverage, the dynamic analysis would report the following *IC_OD* tuples for call site 1:

⟨`A1,A:m,1,Imp1,Imp1:DevM()`⟩
⟨`A1,A:m,1,Imp2,Imp2:DevM()`⟩
⟨`A1,A:m,1,Imp3,Imp3:DevM()`⟩
⟨`A2,A:m,1,Imp1,Imp1:DevM()`⟩
⟨`A2,A:m,1,Imp2,Imp2:DevM()`⟩
⟨`A2,A:m,1,Imp3,Imp3:DevM()`⟩.

The measure of middle granularity, denoted by *IC_OM* in [3] records tuples $\langle C_1, m_1, C_2, m_2\rangle$ dropping the call site index I from the *IC_OD* tuple. The measure of lowest granularity, denoted by *IC_OC* in [3] records tuples $\langle C_1, m_1, C_2\rangle$ dropping the callee method from the *IC_OM* tuple. Clearly, *IC_OD* subsumes *IC_OM* and *IC_OM* subsumes *IC_OC*.

One can use these tuples to compute a measure for each class; for example the *IC_OD* measure for class C is equal to the number of tuples for which the caller method m_1 is a method defined in C. Arisholm et al. [3] show through detailed statistical analysis that the dynamic measures capture new dimensions of complexity of classes compared to traditional coupling measures [8], and predict quality attributes such as changeability better than traditional measures.

For the rest of this paper we will focus on the *IC_OD* measure. Our goal is to design a static analysis that answers the question: given the set of Java classes *Cls* (i.e., the component to be analyzed) what are the *IC_OD* couplings that can be observed throughout all possible executions of arbitrary client code built on top of *Cls*?

Example. Consider the package in Figure 2; this example is based on classes from the standard library `java.text` with some modifications made to better illustrate the problem and our approach. Here *Cls* consists of the classes in Figure 2 plus `String`. Note that classes `SimpleBoundary`, `BreakData`, `WordBreakData`, `LineBreakData` and `CharBreakData` have package visibility and cannot be accessed directly by client code. The remaining classes, `BreakIter`, `CharIter` and `StringCharIter` are designated as accessible classes.

Consider the *IC_OD* tuples in methods of class `SimpleBoundary` (i.e., we are interested in computing the *IC_OD* measure for this class). For brevity we will use `SB` to denote `SimpleBoundary`, `SCI` to denote `StringCharIter`, and `WBD, LBD` and `CBD` to denote `WordBreakData`, `LineBreakData` and `CharBreakData` respectively.

Call site 1 in `setText` triggers coupling ⟨`SB,setText,1,SCI,getBeginIndex`⟩. Similarly, call site 2 in `first` triggers coupling ⟨`SB,first,2,SCI,getBeginIndex`⟩. Finally, call site 3 in `nextPosition` triggers couplings
⟨`SB,nextPosition,3,WBD,WBD:forward`⟩,
⟨`SB,nextPosition,3,CBD,CBD:forward`⟩,
⟨`SB,nextPosition,3,LBD,LBD:forward`⟩.

It is easy to see that for each of these couplings one can write a client of the classes in Figure 2 that will trigger that coupling. For example, client code may call `getWordInstance` (thus initializing the newly created `SimpleBoundary` with `WordBreakData`); subsequently it can access `SimpleBoundary.next` (as `next` is part of the public interface of `BreakIter`) which will trigger the coupling between `SimpleBoundary` and `WordBreakData`.

```
package iter;
public abstract class BreakIter{
   public static BreakIter getWordInstance() {
      return new SimpleBoundary(new WordBreakData()); }
   public static BreakIter getCharInstance() {
      return new SimpleBoundary(new CharBreakData()); }
   public static BreakIter getLineInstance() {
      return new SimpleBoundary(new LineBreakData()); }
   public abstract int first();
   public abstract int next();
   public abstract CharIter getText();
   public abstract void setText(CharIter);
   ... }
final class SimpleBoundary extends BreakIter {
   private CharIter text;
   private BoundaryData data;
   private int pos;
   public void SimpleBoundary(BoundaryData d) {
      text=new StringCharIter(...); data=d; pos=0; }
1  public void setText(CharIter ci) { text=ci; pos=text.getBeginIndex(); }
   public CharIter getText() { return text; }
2  public int first() { pos=text.getBeginIndex(); return pos; }
   public int next() { pos=nextPosition(pos); return pos; }
3  private int nextPosition(int offset) { ...data.forward();... }
   ... }
abstract class BoundaryData { ... }
class WordBreakData extends BoundaryData { public void forward(){...} ... }
class CharBreakData extends BoundaryData { public void forward(){...} ... }
class LineBreakData extends BoundaryData { public void forward(){...} ... }
public interface CharIter { public int getBeginIndex(); }
public StringCharIter implements CharIter {
   public StringCharIter(String s) {...}
   public int getBeginIndex() {...}
   ... }
```

Figure 2: Package `iter`.

3.2 Discussion

We employ the following constraint, standard for other problem definitions that require analysis of incomplete programs [26, 24, 20]. We only consider executions in which the invocation of a boundary method does not leave *Cls*—that is, all of its transitive callees are also in *Cls*. If we consider the possibility of unknown subclasses we may have all instance calls from *Cls* potentially be "redirected" to unknown external code that may affect the coupling inference.

Thus, *Cls* is augmented to include the classes that provide component functionality as well as all other classes transitively referenced. In the experiments presented in Section 5 we included all classes that were transitively referenced by *Cls*. This approach restricts analysis information to the currently "known world"—that is, the information may be invalidated in the future when new subclasses are added to *Cls*. Another approach is to change the analysis to make worst case assumptions for calls that may enter some unknown overriding methods. However, in this case, the analysis will be overly conservative. Thus, we believe that it is more useful to restrict the analysis to the known world; of course, the analysis user must be aware that the information is valid only for the given set of known classes.

4 Fragment Class Analysis for Dynamic Coupling Measures

Class analysis determines the classes of the objects that a given reference variable or a reference field may point to [1, 5, 16, 13, 18, 7, 19, 22, 25, 27, 28]. This information has a wide variety of uses in software tools and optimizing compilers. In this paper, class analysis information is used to approximate the set of *IC_OD* coupling tuples that can happen when arbitrary client code is built on top of *Cls*. This is done by using the class analysis solution to approximate the set of possible classes for the caller, and the set of possible classes for the callee. There is a large body of work on class analysis and related points-to analyses with different trade-offs between cost and precision. In this paper, we consider computation of dynamic coupling measures based on two well known and simple class analyses: Class Hierarchy Analysis (CHA) [12] and Rapid Type Analysis (RTA) [5].

CHA is the simplest form of class analysis. To determine the possible bindings of polymorphic variables and fields CHA examines the structure of the class hierarchy. For a reference variable r or an instance field f of declared type C, the set of possible classes for the objects of r is the set containing C and all direct and transitive subclasses of C (excluding abstract classes). For example, the possible classes for `data` in `nextPosition` in Figure 2 are `WordBreakData`, `LineBreakData` and `CharBreakData`. An implementation of CHA typically assumes that there is a whole-program—that is, there is a `main` method which is the start of the program execution; it starts at `main` and maintains a set of reachable methods R. Whenever a reachable method is found, CHA examines the call sites in this method. For each call site $l = r.m(...)$, it approximates the possible run-time classes for r based on the hierarchy and for each C, it finds the appropriate run-time target m_j based on C and the compile-time target m. Each such m_j is added to R.

RTA is another simple form of class analysis. It improves on CHA by taking into account what classes are instantiated in the program. RTA assumes a whole program and starts from `main` as the first reachable method. Whenever a reachable method is found, RTA examines the call sites *and* the instantiation sites in this method. For each call site $l = r.m(...)$, it approximates the possible run-time classes for r based on the hierarchy and for each C finds the appropriate run-time target m_j. If C has been instantiated, m_j is added to R; otherwise, m_j is added to a set of potential targets that would become reachable if C is instantiated. At instantiation sites RTA records the class that has been instantiated and adds to R all potential targets associated with the instantiated class. To summarize, the RTA solution for a reference variable or a reference object field is the set of *instantiated* classes compatible with its declared type.

4.1 Fragment Class Analysis

Class analyses are typically designed as *whole-program analyses*; they take as input a complete program and produce a class analysis solution that reflects bindings throughout the entire program. However, the problem considered in this paper requires class analysis of a partial program. The input is a set of classes *Cls* and the analysis needs to approximate dynamic couplings that could happen across possible executions of arbitrary client code built on top of *Cls*. To address this problem we make use of a general technique called *fragment analysis* due to Nasko Rountev [23, 26, 24]. Fragment analysis works on a program fragment rather than on a complete program; in our case the fragment is the set of classes *Cls*.

Initially, the fragment analysis produces an artificial `main` method that serves as a placeholder for client code written on top of *Cls*. Intuitively, the artificial `main` simulates the possible flow of classes between *Cls* and the client code. Subsequently, the fragment analysis attaches `main` to *Cls* and uses some whole-program class analysis engine to compute class analysis information which summarizes the possible effects of arbitrary client code. The fragment analysis approach can be used with a wide variety of class analyses [1, 5, 16, 13, 18, 7, 19, 22, 25, 27, 28]; for the purposes of this

```
void main() {
  BreakIter ph_BI;
  CharIter ph_CI;
  StringCharIter ph_SCI;
  String ph_string = "string literal";
  ph_BI = BreakIter.getWordInstance();
  ph_BI = BreakIter.getCharInstance();
  ph_BI = BreakIter.getLineInstance();
  ph_BI.setText(ph_CI);
  ph_CI = ph_BI.getText();
  ph_BI.first();
  ph_BI.next();
  ph_CI.getBeginIndex();
  ph_SCI = new StringCharIter(ph_string);
  ph_CI = ph_SCI; }
```

Figure 3: Placeholder `main` method for `iter`.

paper we only consider fragment analysis used with CHA and RTA.

The placeholder `main` method for the classes from Figure 2 is shown in Figure 3. The method contains variables for types from *Cls* that can be accessed by client code. The statements represent different possible interactions involving *Cls*; their order is irrelevant because the whole-program analysis is flow-insensitive. Method `main` invokes all public methods from the classes in *Cls* designated as accessible. The last statement handles possible assignment conversions between `CharIter` and `StringCharIter`.

The details of the fragment analysis will not be discussed here; they can be found in [26]. For the purposes of our analysis we discuss the following property of the results computed by the fragment analysis, originally outlined in [24]. Consider some client program built on top of *Cls* and an execution of this program (the program must satisfy the constraints discussed in Section 3.2). Let r be a variable or a reference field declared in *Cls* and at some point during execution r refers to a heap object of class C. In the fragment analysis solution, class C will be in the set of classes for r. A similar property holds for variables r declared outside of *Cls*. In this case, in the fragment analysis solution, the set for the variable in `main` that has the same type as r will contain C.

4.2 Computation of Dynamic Coupling Measures

input *Methods*: set of reachable methods
Stmt: set of statements in *Methods*
Cs: $Vars \rightarrow \mathcal{P}(Classes)$
output *Dyn*: $Cls \times Methods \times I \times Cls \times Methods$
[1] **foreach** virtual call s_i: $r.m(...)$ s.t. $r \neq$ `this`
[2] **if** s_i enclosed in static method C_1:n
[3] add $\{\langle C_1, n, i, C_2, target(C_2, m)\rangle \mid C_2 \in Cs(r) \wedge C_1 \neq C_2\}$ to *Dyn*
[4] **else if** s_i enclosed in instance method n (n can be constructor)
[5] add $\{\langle C_1, n, i, C_2, target(C_2, m)\rangle \mid C_1 \in Cs(this_n) \wedge C_2 \in Cs(r) \wedge C_1 \neq C_2\}$ to *Dyn*
[6] **foreach** static call s_i: $C_2.m(...)$
[7] **if** s_i enclosed in static method C_1:n, $C_1 \neq C_2$
[8] add $\langle C_1, n, i, C_2, m\rangle$ to *Dyn*
[9] **else if** s_i enclosed in instance method n (n can be constructor)
[10] add $\{\langle C_1, n, i, C_2, m\rangle \mid C_1 \in Cs(this_n) \wedge C_1 \neq C_2\}$ to *Dyn*

Figure 4: Computation of dynamic couplings.

The algorithm in Figure 4 computes *Dyn*, the set of tuples $\langle C_1, n, i, C_2, m\rangle$ approximating the possible dynamic couplings when arbitrary client code is written on top of *Cls*. The algorithm is parameterized by class analysis, *Cs*. The precision of the computation of the dynamic measures depends on the precision of the underlying class analysis. If the class analysis is more precise (i.e., the solution contains fewer classes per reference variable), then the set of couplings will be more precise (i.e., fewer coupling tuples will be reported).

We briefly discuss the processing of virtual calls (lines 1-5). Lines 2-3 address the case when virtual call s_i: $r.m(...)$ is enclosed by a static method n declared in class C_1. The analysis adds to *Dyn* a dynamic tuple $\langle C_1, n, i, C_2, target(C_2, m)\rangle$ for each class C_2 that is a possible run-time class for r according to the class analysis; $target(C_2, m)$ finds the run-time target method based on class C_2 and compile-time target m. Lines 4-5 address the case when the virtual call is enclosed by an instance method n. In order to approximate the possible dynamic couplings through virtual call s_i: $r.m(...)$ the analysis considers

the possible run-time classes of the implicit parameter $\texttt{this}_n$ of the enclosing method, and the possible run-time classes for r. Consider the example in Figure 1. Suppose that the class analysis computes that $Cs(\texttt{this}_{\texttt{A:m}})=\{\texttt{A1,A2}\}$ and $Cs(\texttt{imp})=\{\texttt{Imp1,Imp2,Imp3}\}$ and consider the couplings that result from call `imp.DevM()`. The set of classes for $\texttt{this}_{\texttt{A:m}}$ approximates the concrete receivers of method `A:m` which in this case are `A1` and `A2`. The set of classes for `imp` approximates the possible callee classes, which in this case is {`Imp1,Imp2,Imp3`}. The analysis outputs the six *IC_OD* tuples outlined in Section 3.1.

5 Experimental Study

The goal of the experimental study is to address the following question. How *imprecise* the analysis is when instantiated with CHA and with RTA—that is, how often it reports infeasible coupling tuples?

For the experiments we used several Java components from the standard library packages `java.text` and `java.util.zip` (used in related analyses [24], [20] and [21]). The components are described briefly in the first three columns of Table 1. Each component contains the set of classes in *Cls* (i.e., the classes that provide component functionality plus all other classes that are directly or transitively referenced); the number of classes in *Cls* and the number of functionality classes is shown in column (3). We considered the *IC_OD*, *IC_OM* and *IC_OC* tuples for the classes that provide component functionality. For the rest of this section we primarily focus on the *IC_OD* tuples as the other kinds of tuples are trivially derived from them. In particular we consider all tuples $\langle C_1, m_1, I, C_2, m_2 \rangle$ where m_1 is a method reachable by RTA, and both C_1 and C_2 are classes that provide component functionality (i.e., we exclude couplings of functionality classes to library classes). The standard library classes are typically well-tested and do not affect the quality attributes assessed by the coupling measures (e.g., they do not contribute to the fault-proneness or changeability of the functionality classes). As mentioned earlier, we instantiate the algorithm in Figure 4 with CHA and RTA.

5.1 Results

Columns (4) and (5) in Table 1 show the number of *IC_OD*, *IC_OM* and *IC_OC* tuples computed by the analysis in Figure 4 when instantiated with CHA and RTA respectively. Clearly, RTA improves precision over CHA. The different granularity level of the three object coupling measures is evident from these results; for most components *IC_OD* is substantially larger than *IC_OM* and *IC_OM* is substantially larger than *IC_OC*. This reflects the structure of the code. It happens often that within the same method, the same virtual call (i.e., a call to the same method with the same receiver class) appears several times at different call sites; when the call site is dropped from the tuple, previously distinct tuples are treated as one and thus there is a drop from *IC_OD* to *IC_OM*. Also, it happens often that within the same method, there are calls to distinct methods with the same receiver class; when the callee is dropped from the tuple, distinct tuples are treated as one and this accounts for the drop from *IC_OM* to *IC_OC*.

We additionally categorize the *IC_OD* tuples into *monomorphic* and *polymorphic*. The monomorphic tuples result from call sites that are resolved uniquely by CHA—that is, both the caller class and the callee class are unique according to CHA. For example, if a static method contains virtual call s_i: $r.m()$ and there is only one possible receiver class for r according to CHA, the corresponding *IC_OD* tuple is categorized as monomorphic. The polymorphic tuples are due to call sites for which either the caller or the callee resolve to more than one class according to CHA. Polymorphic tuples are further categorized as (i) polymorphic only in the caller, (ii) polymorphic only in the callee, and (iii) polymorphic both in the caller and the callee. For example, the six tuples that result due to call site $imp.DevM()$ in Figure 1 are categorized as polymorphic in both the caller and the callee. There is polymorphism in the caller because the class analysis determines two possible classes for implicit parameter `this` of method `A:m`, namely `A1` and `A2`. Similarly, there is polymorphism in the callee because the

(1)Component	(2)Functionality	(3)#Class in *Cls*/ #Functionality	(4)CHA-based Analysis			(5)RTA-based Analysis		
			IC_OD	*IC_OM*	*IC_OC*	*IC_OD*	*IC_OM*	*IC_OC*
`gzip`	GZIP IO streams	199/6	26	19	9	**8**	**7**	**4**
`zip`	ZIP IO streams	194/6	29	8	7	24	4	4
`checked`	IO streams&checksums	189/4	4	4	2	4	4	2
`collator`	text collation	203/15	134	75	36	130	71	32
`date`	date formatting	205/17	358	223	99	335	205	92
`number`	number formatting	198/10	120	55	19	113	52	16
`boundary`	iteration over boundaries	193/13	239	56	27	230	47	25

Table 1: Java components and statically inferred dynamic coupling tuples.

Component	#MONO	#POLY			
		Caller	Callee	Both	Total
`gzip`	2	0	4	**2**	**6**
`zip`	24	0	0	0	0
`checked`	4	0	0	0	0
`collator`	107	0	23	0	23
`date`	171	3	161	0	164
`number`	98	0	15	0	15
`boundary`	182	0	48	0	48

Table 2: Dynamic tuples category statistic.

Component	Polymorphic Tuples		
	CHA-based	RTA-based	Actual
`gzip`	**24**	**6**	**5**
`zip`	5	0	0
`checked`	0	0	0
`collator`	27	23	23
`date`	187	164	164
`number`	22	15	15
`boundary`	57	48	48

Table 3: Statically inferred polymorphic tuples.

class analysis determines three possible classes for `imp`, namely `Imp1`, `Imp2` and `Imp3`.

Table 2 shows the results of this categorization for the RTA *IC_OD* tuples (given in column 5 of Table 1). Most of the polymorphic tuples are callee-polymorphic; however, caller-polymorphic tuples still exist and it is important that an analysis for the computation of dynamic coupling measures considers both polymorphism in the caller and in the callee. Table 2 underscores the importance of measures that take into account polymorphism—there is a large number of polymorphic tuples and most of these tuples would have been omitted with traditional coupling measures. Therefore, in order to assess code quality attributes appropriately, it is essential to consider dynamic coupling measures.

5.2 Analysis Precision

The issue of analysis precision is important for the static analysis for the computation of dynamic coupling measures. If the analysis is imprecise it may report coupling tuples that cannot happen for any execution of the program. Such information is misleading as it may overestimate the complexity of a class, improperly assess the quality attributes of the code and ultimately undermine the usefulness of the dynamic coupling measures.

We examined the *IC_OD* tuples computed by the two instantiations of our analysis and for each tuple we attempted to write client code that would exhibit that tuple. We were able to prove that all monomorphic tuples in our code base are feasible (recall that for comparison purposes we consider tuples in methods reachable by RTA).

Table 3 shows the number of polymorphic tuples: the number for the CHA-based analysis, the number for the RTA-based analysis and the actual number obtained by manual inspection (i.e., after finding a client whose execution on top of *Cls* would exhibit the tuple). Table 3 shows that the RTA-based analysis achieves almost perfect precision as all tuples but one can be exercised by a client (the results for `gzip` are shown in boldface in Tables 3,1 and 2 to underscore this imprecision). The CHA-based analysis, although not as precise as the RTA-based analysis, is still close to the actual result in most cases.

5.3 Conclusions

The static analyses considered in this paper are simple and easy to implement; they have practical cost, practically linear in the size of the program. Utilizing these analyses can lead to efficient computation of dynamic coupling measures in practice. In addition, the analysis based on RTA achieves almost perfect precision and thus it may present a viable, more convenient and more practical alternative to dynamic analysis for the purposes of the computation of dynamic coupling measures. Of course, a threat to the validity of these results is the relatively small code base; clearly, the results need to be confirmed on additional code bases.

6 Related Work

There is a lot of work on coupling measures [10, 11, 8, 9, 30, 29, 17]. These coupling measures are typically computed by code analysis and do not take into account polymorphism, thus underestimating code complexity. Our work is based on static analysis as well, but the measures we compute specifically account for polymorphism.

Our work is related to work on metrics for polymorphism. Eder et al. define a set of coupling measures that consider polymorphism [14]; however, they do not consider the computation of these measures. In contrast our work focuses on the computation of coupling measures; it provides a static analysis algorithm that captures the measures defined in [3]. Benlarbi and Melo define and empirically investigate the quality impact of polymorphism on object-oriented design [6]. They define measures of polymorphic behavior for C++ as follows: for each class C, they consider the number of polymorphic (i.e., virtual) methods that appear in C and its ancestors and descendants. Similarly to the work by Arisholm et al. [3] their statistical analysis shows that such polymorphism affects quality attributes such as the fault-proneness of class C. This work additionally underscores the need to consider polymorphism in measures. The measures by Benlarbi and Melo [6] capture a different dimension of polymorphism compared to the measures by Arisholm et al. [3], when considering polymorphism in server classes. Intuitively, they capture the complexity due to polymorphism in the hierarchy (i.e., the complexity of polymorphic calls through `this` such as `this.m()`) while the measures from [3] aim to capture non-inheritance couplings (i.e., the complexity of calls not through `this`). Clearly, the two measures complement each other.

7 Conclusions

In this paper, we have proposed a new approach to the computation dynamic coupling measures by utilizing static class analysis. The main contributions of our work are the following. First, we propose a static analysis framework for the computation of dynamic coupling measures for strictly-typed object-oriented languages such as Java; our analysis is parameterized by a class analysis and works on incomplete programs. Second, we present an empirical investigation that indicates that our analysis achieves almost perfect precision. Therefore, we believe that our approach presents a viable alternative to the more expensive and complex dynamic analysis for the purposes of computation of dynamic coupling measures.

In the future we plan to experiment with additional components as well as complete programs. Also, we plan to experiment with other instances of our framework that use more expensive and precise class analyses such as points-to analysis. Finally, we plan to extend our methodology to support analysis of dynamically-typed object-oriented languages.

Acknowledgements

The authors would like to thank Nasko Rountev for providing the component data, and the CASCON 2006 reviewers whose comments helped improve this paper.

About the Authors

Yin Liu is a PhD student at Rensselaer Polytechnic Institute in Troy, New York. Ana Milanova is an assistant professor at Rensselaer Polytechnic Institute, Troy, New York.

References

[1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming*, pages 2–26, 1995.

[2] E. Arisholm. Dynamic coupling measurement for object-oriented software. In *IEEE METRICS*, pages 33–42, 2002.

[3] E. Arisholm, L. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Software Engineering*, 30(8):491–506, 2004.

[4] E. Arisholm, D. Sjoberg, and M. Jorgensen. Assessing the chaneability of two object-oriented design alternatives—a controlled experiment. *Emperical Software Engineering*, 6(3):231–277, 2001.

[5] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[6] S. Benlarbi and W. Melo. Polymorphism measures for early risk prediction. In *ACM/IEEE International Conference on Software Engineering*, pages 334–344, 1999.

[7] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDD's. In *ACM Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[8] L. C. Briand, P. T. Devanbu, and W. L. Melo. An investigation into coupling measures for C++. In *ACM/IEEE International Conference on Software Engineering*, pages 412–421, 1997.

[9] L. C. Briand, J. Wust, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *IEEE International Conference on Software Maintenance*, pages 475–482, 1999.

[10] S. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 197–211, 1991.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, 1994.

[12] J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[13] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *ACM Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[14] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. In *Conference on Information and Knowledge Management*, 1992.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[16] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, 1997.

[17] T. Gyimoty, R. Ferenc, and I. Siket. Emprirical validation of object-oriented metrics on open source software for fault prediciotn. *IEEE Trans. Software Engineering*, 31(10):897–910, 2005.

[18] O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, pages 153–169, 2003.

[19] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses

for Java. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, 2001.

[20] A. Milanova. Precise identification of composition relationships for UML class diagrams. In *IEEE/ACM Conference on Automated Software Engineering*, pages 76–85, 2005.

[21] A. Milanova. Composition inference for UML class diagrams. Technical Report 06-06, Rensselaer Polytechnic Institute, April 2006.

[22] A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *ACM International Symposium on Software Testing and Analysis*, pages 1–12, 2002.

[23] A. Rountev. *Dataflow Analysis of Software Fragments*. PhD thesis, Rutgers University, 2002.

[24] A. Rountev. Precise identification of side-effect free methods. In *IEEE International Conference on Software Maintenance*, pages 82–91, 2004.

[25] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, 2001.

[26] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java softwhare. *IEEE Trans. Software Engineering*, 30(6):372–386, June 2004.

[27] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an emprirical comparison. Technical report, U. Passau, September 2000.

[28] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, 2000.

[29] F. G. Wilkie and B. Kitchenham. Coupling measures and change ripples in C++ application software. *Journal of Systems and Software*, 52(2):157–164, 2000.

[30] P. Yu, T. Systa, and H. A. Muller. Predicting fault-proneness using oo metrics: An industrial case study. In *European Conference on Software Maintenance and Reengineering*, pages 99–107, 2002.