Deriving Coupling Metrics from Call Graphs

Simon Allier*†, Stéphane Vaucher*, Bruno Dufour*, and Houari Sahraoui*
†VALORIA, Université de Bretagne-Sud
*DIRO, Université de Montréal
{alliersi, vauchers, dufour, sahraoui}@iro.umontreal.ca

***Abstract*—Coupling metrics play an important role in empirical software engineering research as well as in industrial measurement programs. The existing coupling metrics have usually been defined in a way that they can be computed from a static analysis of the source code. However, modern programs extensively use dynamic language features such as polymorphism and dynamic class loading that are difficult to capture by static analysis. Consequently, the derived metric values might not accurately reflect the state of a program. In this paper, we express existing definitions of coupling metrics using call graphs. We then compare the results of four different call graph construction algorithms with standard tool implementations of these metrics in an empirical study. Our results show important variations in coupling between standard and call graph-based calculations due to the support of dynamic features.**

I. Introduction

Many studies have demonstrated the importance of software metrics and measurement programs to quantitatively evaluate and improve the quality of software products [10], [7], [14]. In these measurement programs, metrics of internal attributes of software products such as size and coupling are used to assess quality factors such as error proneness, changeability, and reusability. To allow metric gathering, measurement tools have been proposed either as standalone software or as parts of integrated development environments [8].

Most object-oriented metrics are intended to be computed static by examining the source code of a software system. These metrics, however, were defined in the nineties to manage the quality of systems developed in C++ [6]. Since then, programming languages as well as development practices have evolved significantly: modern programs often make an extensive use of dynamic language features such as polymorphism, dynamic class loading, dynamic class generation, and reflection. In comparison, these features were either used infrequently in C++, or were simply not available.

When computing metrics, it is important to account for dynamic features used by a system in order to obtain an accurate portrait of the behavior of an application. In particular, coupling metrics try to quantify the amount of interaction between different classes in a system. Variations in the strategies used to handle the dynamic features of a program can therefore significantly affect coupling measures. For example, without proper handling of polymorphic method invocation and dynamic class loading, one could miss an important portion of the actual coupling between classes or, conversely, overestimate it. While some coupling metric definitions consider polymorphism (*e.g.,* [5]), the vast majority of tool implementations disregard it completely. Other dynamic features, such as dynamic class loading, are ignored in both metric definitions and tool implementations. This is in part due to the common belief that computing the required information statically is too difficult, time-consuming or imprecise. Moreover, even when polymorphism is taken into account, the level of precision used to determine the set of potentially invoked methods can still greatly influence the computed metric results.

The objective of this paper is to investigate the effect of various strategies to handle dynamic language features on the statically computed coupling metric values. This information is necessary to allow metric tool implementors to make informed design decisions. Specifically, we focus on two particularly widespread dynamic features: polymorphism and dynamic class loading. This paper therefore makes the following contributions:

- We present a formulation of existing coupling metric definitions that relies on call graphs to capture the behavior of program related to polymorphism and dynamic class loading.
- We use type-analysis algorithms combined with programming heuristics to build call graphs that approximate the runtime behaviour of a program with various levels of precision. These graphs are then used to quantify the coupling.
- We conduct an empirical study of the effect of polymorphism and dynamic class loading on coupling using two well-known systems. Our results show that classical methods under- or overestimate the coupling for systems where dynamic features are used. In some situations, the variations are very significant.

The remainder of this paper is organized as follows. Section II gives an overview of the related work. Section III describes our approach. After giving formal definitions of some coupling metrics, we show how polymorphism and dynamic class loading are considered when building call graphs. We then illustrate the metric calculation from these call graphs. A case study is discussed in Section IV. Finally, conclusive remarks are given in Section V.

II. Related Work

Over the past two decades, numerous studies have established relationships between product metrics and quality characteristics of object-oriented (OO) systems. Briand and Wüst [4] summarize some of these studies. In particular, metrics are used to quantify internal attributes of software such

as size-complexity, coupling, cohesion and inheritance. The program characteristics that are studied range from reliability to maintainability.

The majority of software metrics are computed statically from either source code or binaries. Dynamic language features, however, are known to be difficult to capture by a purely static analysis [20]. To circumvent this problem, metrics have been proposed to measure the actual coupling between objects at runtime using a dynamic analysis. For each execution, precise coupling metric values can be computed. Arisholm *et al.* [1] defined a suite of dynamic coupling metrics at the class and object levels for the problem of change proneness evaluation during the evolution of software. All their metrics are defined for individual executions. They showed empirically that their metrics are better predictors of code change than the static metrics. Similarly, Yacoub *et al.* [19] proposed a set of metrics to dynamically measure coupling and complexity. Some of the metrics were defined for single executions, others were general to a set of executions with respect to a specific set of scenarios. Each scenario was assigned a probability corresponding to its execution frequency. The final metric value is defined as the mathematical expectation of the individual scenario-based metric values by considering the scenario probabilities. These metrics served later to study the reliability risk at the architecture level [18]. While these dynamic techniques can precisely capture relationships between classes, they suffer from limitations. First, executions of a program with different parameters usually lead to different values for the same metric. Consequently, it is difficult to derive a universal value for a particular metric from execution traces. This problem is compounded when we wish to extract metrics on every module of a large program, as it requires finding traces that ensure an adequate coverage of a program. Requiring an adequate sample of all executions through a program is prohibitively expensive, and consequently, most organisations still heavily rely on static metrics despite their limitations. While our work shares a goal with dynamic metrics, our focus is on investigating the impact of dynamic language features on all executions rather than a finite set of concrete executions.

Static analyses that cannot cope with certain dynamic features such as dynamic class loading and reflection are forced to make conservative assumptions about the possible behavior of a program that often significantly degrade the precision of the analysis. Because the required information is only observable at runtime (and therefore, determining the behavior purely statically is an undecidable problem), techniques have been recently developed that inject various levels of dynamic information in a static analysis in order to improve their precision (*e.g.,* [3]). In particular, such techniques have been applied to the problem of extraction method invocation relationships from programs (in the form of call graphs). A similar approach could be used to extend metric extraction tools to estimate runtime behavior. To the best of our knowledge, this approach remains unexplored.

Other researchers have used program analysis formalisms to express coupling metric computations. For example, Harman *et al.* [12] defined coupling in terms of program slicing. Program slicing is a technique that removes from a program all entities (*e.g.,* statements) that are not relevant for a given computation. For example, Harman *et al.* use slicing to identify variable definitions outside of a module m used by m as well as variables defined in m used elsewhere in the program. This information is used to quantify coupling in an application. Myers and Binkley [13] have performed a large-scale empirical study of slice-based metrics, including the coupling metrics defined by Harman *et al.* , on a large set of C programs. Their computation relies on system-dependence graphs (SDGs) and procedure-dependence graphs (PDGs), which represent control- and flow-dependence, both intra- and interprocedurally. Computing these dependences in the context of modern, more dynamic languages requires similar techniques as those used this work. Therefore, our study is complementary to the work on sliced-based metrics.

III. Approach

To measure the impact of polymorphism and dynamic class loading on metric results, it is important to define the metrics in a way that enables experimentation with different strategies. In this section, we first review the classical definition of two popular coupling metrics (Coupling Between Objects and Response for Class). We then refine these definitions to account for polymorphism and dynamic class loading.

A. Classic coupling metrics

Chidamber and Kemerer [6] have proposed a set of object-oriented metrics including Coubling Between Objects (CBO) and Response for Class (RFC). Informally, the CBO metric aims to measure the amount of interconnectivity between a given class and other classes in the system, while the RFC metric quantifies the number of distinct methods that can be invoked from a given object. The CBO and RFC metrics defined by Chidamber and Kemerer have later been formalized by Briand *et al.* [5]. We review these definitions next.

a) Coupling Between Object (CBO): According to the original definition of CBO by Chidamber and Kemerer, the CBO for a class is a count of the number of other classes to which it is coupled. A class c is said to be *coupled* to a class d if c uses d or d uses c. A class c uses a class d if one of its methods invokes a method defined in class d or accesses a field defined in class d. More formally, Briand *et al.* have defined the CBO metric as follows:

$$CBO(c) = |d \in C - \{c\} | uses(c,d) \vee uses(d,c)|$$

where C is a set of classes and $c \in C$, and

$$uses(c,d) = (\exists m \in M_I(c) : \exists m' \in M_I(d) : m' \in PIM(m))$$
$$\vee (\exists m \in M_I(c) : \exists a \in A_I(d) : a \in AR(m))$$

where $A_I(c)$ is the set of implemented attributes in class c, $AR(m)$ is the set of referenced attributes in the method

m, $M_I(c)$ is the set of implemented methods in class c and $PIM(m)$ is the set of polymorphically invoked methods of m. An attribute a is in $AR(m)$ if a is read or written in the body of the method m.

b) Response For Class (RFC): Chidamber and Kemerer define RFC for a given class as the number of distinct methods that can be (directly) invoked in response to a message to an object of that class. Briand *et al.* have formalized this definition as follows:

$$R_0(c) = M(c)$$

$$R_1(c) = \cup_{m \in M(c)} PIM(m)$$

$$RFC(c) = |R_0(c) \cup R_1(c)|$$

where $M(c)$ is the set of implemented, overridden and inherited methods in class c, and $PIM(m)$ is again the set of polymorphically invoked methods of m.

B. Accounting for dynamic language features

Polymorphism and dynamic class loading can affect the CBO and RFC metric computations in two different ways. First, invoking methods from a class C constitutes a use of C according to the classical definition of coupling. Polymorphism is therefore reflected directly in the PIM set computation. Different strategies for determining the set of possible method targets for a call will result in a different PIM set. Second, dynamic class loading can add classes to the set C of all classes in the system, and therefore potentially indirectly affect the PIM computation as well. We examine different strategies for handling polymorphism and dynamic class loading next.

1) Polymorphism: In order to account for polymorphism in the metric computations, it is necessary to know the set of methods that can be invoked at each call site in the program (*i.e.,* invocation targets). This information is commonly obtained by building a *call graph* for the entire program. A call graph is a directed graph in which nodes represent methods of a program, and edges represent calls between these methods.[1] Call graph construction has been extensively studied in the program analysis community. Many techniques have been developed to compute call graphs both statically from source code and dynamically, using one or more concrete program executions. Because dynamic call graphs only represent a subset of the whole behavior of a program, we only consider static call graph building techniques.

[1] Basic call graphs contain a single node for each method in the program, but more precise (context-sensitive) call graph representation are also possible.

[2] http://www.aivosto.com

[3] http://www.borland.com/us/products/together/index.html

[4] http://www.virtualmachinery.com/products.htm

[5] http://www.powertoolsuk.co.uk

[6] http://www.mccabe.com/iq.htm

[7] http://www.spinellis.gr/sw/ckjm/

[8] http://masu.sourceforge.net

[9] http://www.ptidej.net/downloads/pmart/

TABLE I
CBO IMPLEMENTATION DETAILS FOR DIFFERENT METRIC TOOLS

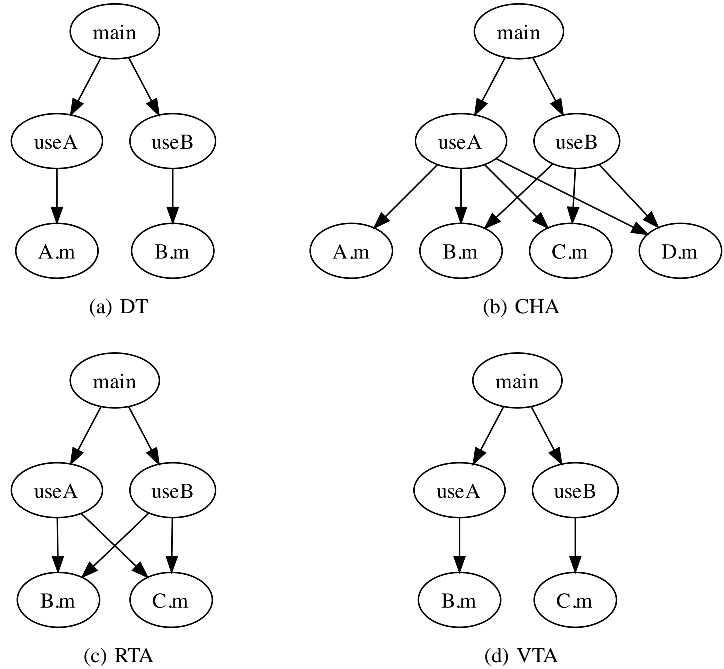
Tool	Type	Considers method invocations?
Aivosto[2]	Commercial	✓, uses declared types
Together[3]	Commercial	✓, uses declared targets
JHawk[4]	Commercial	×, counts referenced types
Powertools[5]	Commercial	×, counts association types
McCabe IQ[6]	Commercial	×, counts external references
CKJM[7]	Research	✓, uses declared targets
MASU[8]	Research	✓, uses declared targets
POM[9]	Research	✓, uses declared targets

Different static call graph building algorithms make different trade-offs in terms of time and precision. Building a precise call graph typically requires sophisticated analyses and large amounts of time and memory, while less precise call graphs can be computed very cheaply. The most basic call graph building algorithm, *Class Hierarchy Analysis* (CHA) [9], only considers the type hierarchy when computing the set of possible targets at a given call site. In other words, CHA assumes that the declared target method and any of its overriding methods could be invoked at a given call site. CHA thus generally overapproximates the set of possible targets methods, but has the advantage of being very inexpensive. *Rapid Type Analysis* (RTA) [2] is almost identical to CHA, but refines its results by considering the set of objects types that may be allocated. The key optimisation of RTA is that it ignores any type not instantiated in the program. This observation is used to prune the set of types computed by CHA and obtain a more precise call graph. *Variable Type Analysis* (VTA) [15] is a simple dataflow analysis that tracks, for each object reference (*e.g.,* variable) in the program, the set of object types that it can contain. This information is used to further reduce the set of possible invocations at any given call site, at the expense of additional computing time. Other, more sophisticated call graph building techniques exist (*e.g.,* [11], [16]), but are not explored in this work because of their high computation costs.

A common strategy in metric extraction tools involves using the declared target of a method call rather than the set of all possible targets to approximate the PIM set. Conceptually, this technique corresponds to building a call graph where an edge is present from method m to method m' if and only if m contains a call with m' as the declared target. While this approach does not account for all possible executions of a program, a survey of existing metric extraction tools indicates that this strategy is used by all tools that consider method invocations as part of their coupling measures. The remaining tools completely ignore method invocations and simply rely on a count of referenced types in a given class as a basis for the coupling measures. Table I presents the results of our survey. Of the eight tools that we examined, none respected the definition provided by Briand *et al.* Because of the popularity of using declared method targets to approximate the PIM sets in practice, we include this algorithm as part of our study for

```
static void main() {
  B b1 = new B();
  C c = new C();
  useA(b1);
  useB(c);
}

static void useA(A a) {
  a.m();
}

static void useB(B b2) {
  b2.m()
}

class A {
  void m() {...}
}

class B extends A {
  void m() {...}
}

class C extends B {
  void m() {...}
}

class D extends B {
  void m() {...}
}
```

Fig. 1. Example for call graph algorithms



(a) DT (b) CHA (c) RTA (d) VTA

Fig. 2. Call graphs for example in Figure 1

comparison purposes. It will henceforth be referred to as *DT*.

a) Example: Consider the example in Figure 1. Method `main` allocates two object instances, one of type `B` and the other of type `C`, before calling methods `useA` and `useB` with these objects as parameters. Manual inspection easily reveals that method `useA` will always result in a call to method `m` defined in class `B`, while a call to method `useB` will always result to a call to method `C.m`. Using the DT call graph building strategy, we obtain the call graph in Figure 2a. Note that this call graph is not a conservative one; it does not include the real runtime behavior of the example program. When using CHA to build the call graph, we obtain a conservative but imprecise call graph. The algorithm considers that the call site `a.m()` in method `useA` can result in calls to all implementations of `m`. Similarly, the algorithm considers that the call `b2.m()` can potentially invoke the implementations of `m` in class `B` and all of its subclasses. The resulting call graph appears in Figure 2b. RTA is able to improve the precision of the resulting call graphs by observing that no objects of types `A` or `D` are ever created. Therefore, it only considers implementations of `m` from classes `B` and `C`. As show in Figure 2c, however, it is not capable of identifying the fact that each polymorphic call site has a single target. Finally, VTA can track the types of the values assigned to variables across method call boundaries, and can successfully compute a precise call graph for the given example (as shown in Figure 2d).

2) Dynamic class loading: Dynamic class loading can also affect a call graph building strategy. Because the specific code that will be loaded at runtime cannot be determined statically, static program analyses are forced to make conservative assumptions regarding the classes that can be loaded dynamically. In this work, we consider that any application (*i.e.,* non-library) class can be potentially loaded at runtime. While this is not guaranteed to be a truly conservative assumption, in practice it is almost always sufficient.

While handling dynamic class loading enables a better measure of the program's behavior, it also introduces a source of imprecision due to the way dynamically loaded classes are typically used in Java. Figure 3 illustrates a common usage scenario. First, a reference to a `java.lang.Class` object is obtained at runtime using the name of the class to load (as a `String`). Second, an instance of the newly loaded class is created using the `Class.newInstance` method. Finally, the object is cast to the appropriate type and used. Because of their conservative nature, call graph building algorithms consider all no-argument constructors as potentially invoked by the call to `newInstance`. This would cause method `foo` to be coupled with every no-argument constructor in all application classes. To solve this problem, we can elide some edges from the call graph following its construction. Specifically, we do not consider any edges originating from call sites that correspond to methods `Class.forName` and `Class.newInstance` when computing the PIM set. For example, for the code in Figure 3, the coupling due to the call to `obj.m()` would be correctly computed.

C. Extracting metrics using a call graph

With the exception of the PIM sets computations, the CBO and RFC values can be easily determined for each class in a system by a shallow analysis of the code. Computing the set $PIM(m)$ is more complex. However, it can be readily

```
void foo() {
  Class c = Class.forName("MyClass");
  MyClass obj = (MyClass)c.newInstance();
  obj.m(); // Use the object
  ...
}
```

Fig. 3. Typical usage of dynamic class loading in Java

computed from a call graph by aggregating the targets for each call site in m. Formally, let a call graph be denoted by $CG(M, E)$ where M is a set of methods in a program and E a set of edges between these methods. We then define $PIM(m)$ as follows:

$$PIM(m) = \{m' | (m, m') \in E \wedge c, d \in C \wedge m \in M_I(c) \wedge m' \in M_I(d)\}$$

In other words, $PIM(m)$ is the union of all method targets for each call site in method m. By varying the call graph building algorithms, we can therefore compute different versions of the coupling metrics that use different strategies to handle dynamic language features.

IV. Case Study

In order to assess the impact of dynamic language features on coupling metric computations, we performed an empirical study using two Java applications: ArgoUML and Azureus. In this section, we describe the experimental setting used, and discuss the empirical results.

A. Experimental setting

a) Applications: ArgoUML 0.18.1 is a UML modelling tool with code generation and reverse-engineering capabilities. It provides the user with a set of views and tools to model programs using UML diagrams, to generate the corresponding code skeletons and to reverse-engineer diagrams from existing code. The set of classes and interfaces C used to compute CBO and RFC corresponds to the elements of the package `org.argouml`. The set C has 1237 classes and 100 interfaces.

Azureus 2.1.0.0 is a popular, multi-platform BitTorrent client that allows users to share files using a peer-to-peer network. This application has 1232 classes and 250 interfaces from the package `org.gudy.azureus2`.

b) Implementation: Our metric computation tool is implemented using Soot, a popular Java static analysis framework [17]. Soot provides the call graph construction algorithms presented in Section III. It also supports dynamic class loading by requiring the user to specify a set of classes that can be loaded at runtime. The tool supports building call graphs from either source code or bytecode. All results from this experiment were computed from compiled bytecode.[10]

We extended Soot to compute the CBO and RFC metrics from call graphs. We also added support for the DT call graph construction algorithm (recall that this algorithm only considers declared targets of calls, and ignores polymorphism). When computing metrics, we only considered coupling between application classes (*i.e.,* excluding the Java standard libraries).

We computed the CBO and RFC metrics for both ArgoUML and Azureus using 4 call graph building algorithms: DT, CHA, RTA, and VTA. In the case of VTA, we used two different versions of the algorithm: one with support for dynamic class loading (henceforth referred to as "VTAd"), and one without. Note that dynamic class loading does not affect the call graphs built using the DT, CHA, and RTA algorithms.

All experiments were performed on an IBM Java Virtual Machine (JVM) version 6.0 running in server mode on an AMD Opteron 2Ghz machine with 8GB of RAM with Fedora Core 7.[11] In order to make the analysis more scalable, we computed the metrics using version 1.4 of the Java standard libraries (rather than 1.6). Because the metric computations, only consider application classes, there is no impact of the final results.

B. Execution times

In order to compare the relative costs of the various algorithms, we computed their execution times for both applications. Table II shows the execution times by call graph algorithm used. The table lists three times for each application: the time required to build the call graph ("CG"), the time required to compute the metrics ("Metrics") from the previously computed call graph, and finally the total of the two previous values[12]. Because both the CBO and RFC metrics are computed simultaneously, we only report combined timings for both metrics. Also note that while the DT algorithm can be conceptually expressed in terms of a call graph, it does not however require an explicit call graph to be built in order to compute the metrics. We therefore use this optimized approach and report a CG computing time of zero for DT.

TABLE II
Execution times (in minutes:seconds)

CG algorithm	ArgoUML			Azureus		
	CG	Metrics	Total	CG	Metrics	Total
DT	0:00	0:49	0:49	0:00	0:48	0:48
CHA	5:11	3:59	9:10	3:15	2:28	5:43
RTA	35:43	4:03	39:46	23:46	2:21	26:07
VTA	12:42	2:31	15:13	7:30	0:50	8:20
VTAd	14:47	2:55	17:42	11:44	1:28	13:12

Table II shows that as expected, the DT algorithm is the fastest, completing in under a minute for each application. While all algorithms complete in under 40 minutes, RTA is the only algorithm that requires longer than 20 minutes to complete. This result is surprising, and we believe it is due to its implementation in Soot. With the exception of RTA,

[10]We also computed the same results from source code for comparison, and as expected found virtually no difference between the two sets of results.

[11]We ran the JVM in compressed reference mode in order to increase the memory efficiency on the 64-bit architecture.

[12]We excluded the time taken to perform common operations such as loading the classes from disk or outputting the metric results.

the algorithms that take polymorphism into account become progressively slower with increased precision. For instance, CHA completes in just under 10 minutes for ArgoUML, while the more precise VTA algorithm with dynamic class loading (VTAd) requires nearly twice the amount of time to run. This difference is entirely due to an increase in call graph building time. In fact, the metric computation time tends to *decrease* with increases in the precision of the call graph building algorithm. Because metric computations only consider *reachable* classes (*i.e.,* classes that have at least one of their methods reachable from the program entry point through application classes), an increase in call graph precision results in less code being examined to compute the metrics. Note that the DT algorithm, however, considers all classes, whether or not they are reachable. In addition, the variant of VTA that does not consider dynamic class loading ignores certain reachable classes that are only accessed via dynamic loading. As a result, it considers even less code that VTAd, which also results in a faster metric computation time (but produces incomplete results). The impact of unreachable code will be examined in more details in Section IV-D.

TABLE III
CALL GRAPH SIZES

CG algorithm	ArgoUML		Azureus	
	Nodes	Edges	Nodes	Edges
CHA	36 872	1 113 377	27 825	384 330
RTA	36 642	1 102 549	27 749	383 650
VTA	32 085	715 109	25 377	279 392
VTAd	36 632	1 858 348	27 076	613 025

Table III shows the size of the call graphs computed by each algorithm. Note that as mentioned above, the DT algorithm does not build an explicit call graph and therefore no size results are provided for it. The table shows that the main impact of the various algorithms is on the number of edges in the graphs rather than the nodes. This shows that most code in an application is potentially reachable, and the improved algorithms only help to disambiguate calls rather than demonstrate that some methods cannot be invoked. Also, the large increase in the number of edges in VTAd clearly demonstrates the importance of properly handling dynamic class loading in practice. Note that VTA produces much smaller call graphs than the other algorithms because it considers an incomplete program. This demonstrates the importance of dynamic class loading in both benchmarks.

C. Distribution of coupling values

In order to study the impact of polymorphism and dynamic class loading on coupling metrics, we computed the frequency of CBO and RFC values in terms of number of classes. Figure 4 shows this frequency for the CBO metric across all call graph building algorithms. Note that the figure uses a logarithmic scale on the horizontal axis in order to reduce the large range of obtained values.[13] The results show that the choice of algorithm has a significant impact on CBO values for ArgoUML. This indicates that ArgoUML makes an extensive use of non-trivial polymorphism (*i.e.,* it features a deep class hierarchy). The results also show that the DT algorithm underapproximates the CBO value for a large number of classes. This is explained by the fact that for truly polymorphic call sites, the class declaring the caller method should be coupled to each callee's class. Examining the differences in CBO values between the DT and VTAd algorithms shows that this is indeed the case. We found that for classes whose CBO value increased by going from DT to VTAd in ArgoUML, the increase in CBO value was on average 13. This clearly shows the importance of polymorphism in this benchmark. Note that even in the cases where CBO and VTAd result in the same CBO value, the set of coupled classes are not necessarily identical. For example, consider once again the code from Figure 1. The contribution to CBO from method `useA` would be 1 according to both DT and VTA. However, in the former case this coupling would be due to class `A` while in the latter case it would be due to class `B`. On the other hand, there are cases where DT *overapproximates* the coupling for a class. For example, in ArgoUML, the class `UmlDiagramRenderer` is coupled to 63 other classes according to DT but only 6 classes according to VTAd. `UmlDiagramRenderer` is an abstract class that provides default implementations of two public methods, which make calls that increase the coupling of the `UmlDiagramRenderer` class to other application classes. The two public methods are however redefined in all subclasses of `UmlDiagramRender`, and therefore cannot be called in practice. VTAd correctly identifies this situation and as a result computes a much lower CBO value for the same class. In fact, even the less precise CHA is sufficient to achieve the same result in this case.

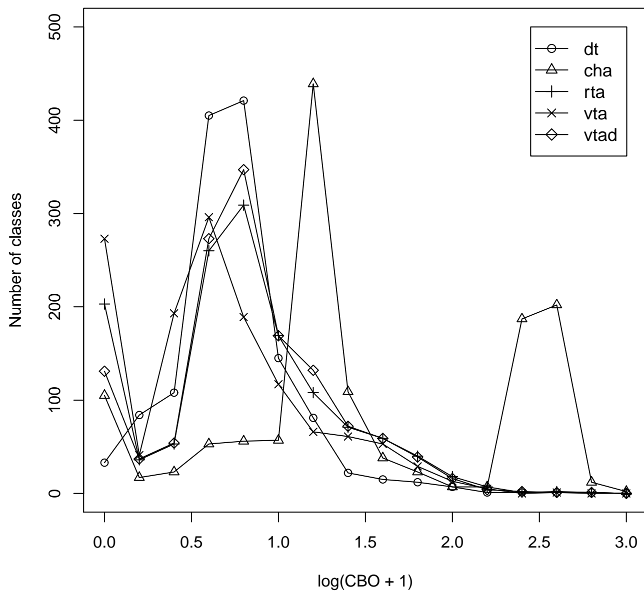
Figure 4b shows that there are much fewer differences between various algorithms in the case of Azureus. This is due to the fact that Azureus uses very little inheritance, and thus very few of its call sites are truly polymorphic. The DT algorithm, however, differs significantly from the other algorithms. In particular, DT reports that only 49 classes are unreachable (CBO=0), whereas the other algorithms identify between 264 and 291 such classes. This constitutes an overapproximation over the CBO value of around 4 on average, and therefore leads to the peaks seen in Figure 4b around 0.2 and 0.6.

Figure 5 shows the frequency of RFC values in terms of number of classes (again using a logarithmic scale). The results are very similar to those obtained for CBO. ArgoUML exhibits a wide variation in RFC results with different call graph algorithms, with DT being biased towards smaller metric values and CHA producing significanly higher RFC values. Azureus exhibits even less variations than with CBO, thus supporting the claim that it makes very little use of deep class hierarchies and true polymorphism.

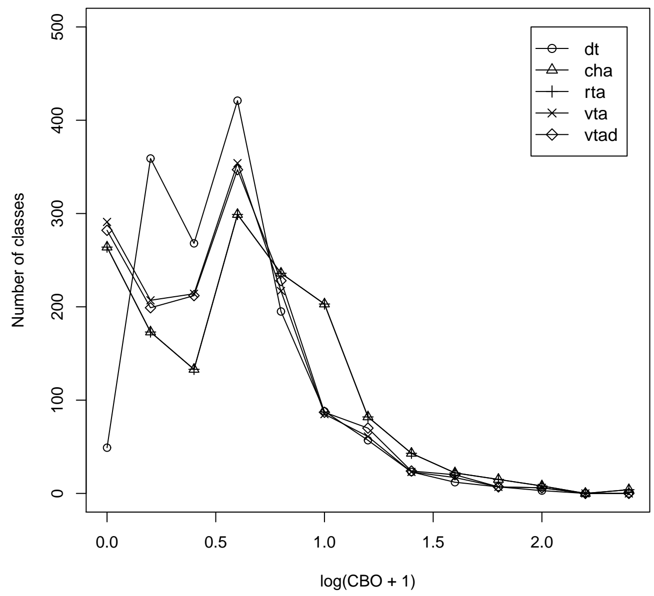
D. Dead code

An important advantage of using call graphs to compute coupling metrics is its capacity to ignore *dead code* (*i.e.,*

[13]We added one to CBO metric values before taking the logarithm in order to avoid problems with classes whose CBO values are zero.

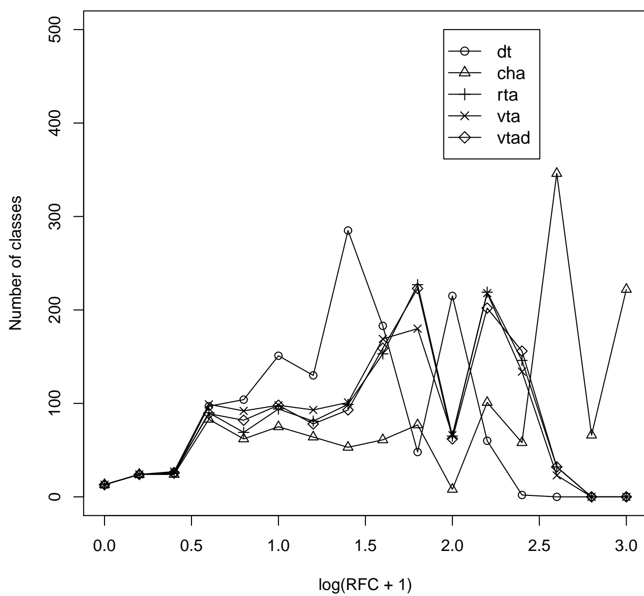


(a) ArgoUML

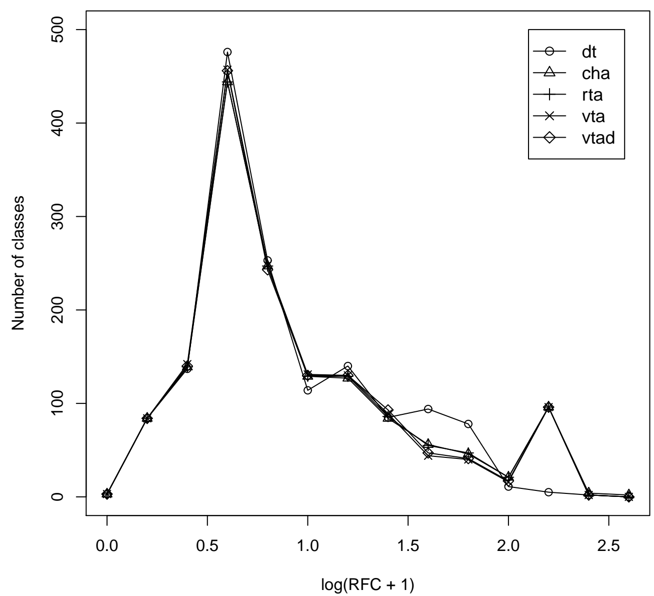


(b) Azureus

Fig. 4. The distribution of CBO



(a) ArgoUML



(b) Azureus

Fig. 5. The distribution of RFC

code that cannot possibly be executed for a given program). The amount of dead code varies with the particular algorithm used to compute the call graph. Conservative algorithms like CHA and VTAd can underestimate the amount of dead code: they can consider a class to be *live* (not dead) when it is not used in practice. Unsafe algorithms like DT can both underapproximate and overapproximate the amount of dead code in a program. Underapproximations are possible, for example, when non-trivial polymorphism is used. With DT, a virtual call with an abstract declared target will contribute to the coupling of the abstract class, while in fact at runtime all calls will necessarily invoke methods in subclasses of this abstract class. Overapproximations, on the other hand, can be caused by the reverse situation. For example, class `ModeCreateLink` in ArgoUML only uses library methods (and therefore is not coupled to any other application class through outgoing invocations), and there are no invocations in the program that use this class as the declared target. Therefore, the `ModeCreateLink` class appears to be dead from the point of view of DT, but VTAd correctly identifies that methods in this class are reachable through polymorphic calls.

Figure 4a clearly shows that the amount of dead code varies with the call graph algorithm used to compute the metrics. Dead classes correspond to those classes whose computed CBO value is 0.[14] Interestingly, the very conservative nature of the CHA algorithm renders it unable to identify most of the dead code in ArgoUML. DT identifies 33 classes as dead (CBO=0), but some of these classes are misclassified. VTAd identifies almost four times this number with 131 dead classes. Note that the 273 classes identified as dead by VTA are not trully unreachable; most of these classes are only reachable through dynamic class loading and reflection. For instance, VTA identifies all 11 concrete subclasses of the abstract `Wizard` class in ArgoUML as dead because they are all loaded and instantiated exclusively via reflection mechanisms. Figure 4b shows a similar trend for dead code in Azureus. Because Azureus has more statically resolved calls than ArgoUML, the variation between the amount of dead code identified by the various algorithms is much smaller than for ArgoUML. For instance, VTAd identifies 282 dead classes, compared to 291 classes for VTA, 279 for RTA and 264 for CHA. Only DT differs significantly from the other algorithms: it only labels 49 classes as unreachable.

Note that for RFC, as shown in Figure 5, there are no classes with RFC of values of 0 with any algorithm. This is due to the fact that each class contains at least a default constructor. The minimum possible RFC value is therefore 1.

E. Interfaces

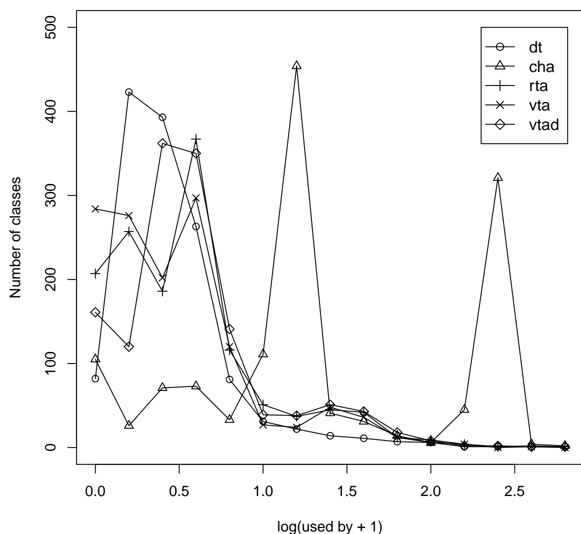
Similarly to dead code, the importance of interfaces in coupling metric computations varies greatly with the various call graph building algorithms used. Because the code itself can use interface methods as declared call targets, the DT algorithm will consider interfaces when performing coupling computations. However, because call graphs resolve polymorphic calls to interfaces, the other algorithms will not include interfaces in their CBO computations. In some cases, this can lead to extreme variations in CBO values. For example, class `NSUMLModelFacade` from ArgoUML has a CBO value of 4 with the DT algorithm because it is never called directly using its static type, but rather invoke through a more generic `Facade` interface. As a result, its computed CBO value with VTAd is 576!

F. Polymorphism

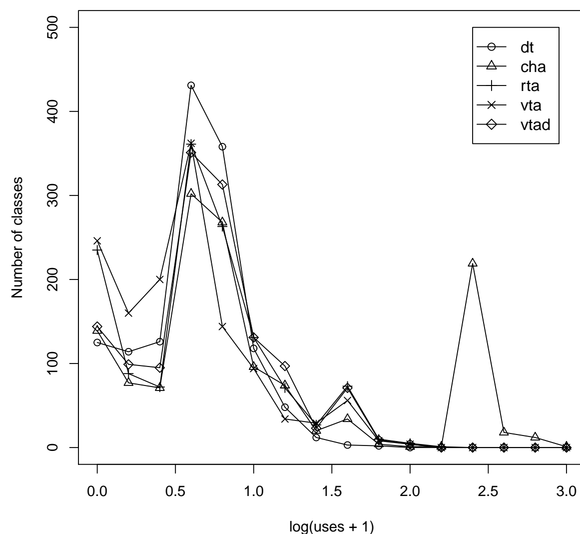
Changes in call graph building algorithms can affect the CBO measure of a class C in two ways: they can vary the set of classes used by *incoming* calls (*i.e.,* calls using methods from C) and by *outgoing* calls (*i.e.,* calls from C to methods in other classes). Therefore, changes in the PIM set of a method m in class C due to a variation in call graph building algorithms most often has an impact on the CBO value of C but also on the CBO values of all classes declaring the potential target method. In order to gain a better understanding of the impact of different algorithms on CBO values, we measured the coupling due to incoming calls (CBO-In) as well as for outgoing calls (CBO-Out). Figure 6 shows the results for ArgoUML. The DT algorithm can once again be seen to underapproximate the coupling for both CBO-In and CBO-Out as compared to VTAd. More interestingly, Figure 6a clearly shows that the overapproximations from CHA mainly stem from CBO-In rather than CBO-Out. This can be explained by the fact that certain classes can be seen as potentially called from a number of program locations based on their inheritance hierarchy, but in reality calls are concentrated towards a small number of specific classes rather than distributed across all potential targets. For example, the class `ActionAddMessagePredecessor` in ArgoUML inherits from `UMLAction`. All calls that use the generic `UMLAction` as a target will contribute to the CBO-In of `ActionAddMessagePredecessor` according to CHA, resulting in a total CBO-In value of 252. VTAd, on the other hand, can determine that very few of these calls actually reach `ActionAddMessagePredecessor`, and as a result assigns a CBO-In value of only 3 to it. In both cases, the CBO-Out value for `ActionAddMessagePredecessor` is 7. Also note that while the incoming calls account for the majority of the differences between algorithms, there is a significant number of classes for which CHA computes a much higher CBO-Out with CHA than other algorithms. This is also due to the inherent imprecision of the CHA algorithm. For these classes, CHA is not able to compute a precise set of targets for some of their call sites.

The results for Azureus (not shown) indicate marginal differences across the various algorithms for both CBO-In and CBO-Out. This is again due to the fact that most calls in this application are non-polymorphic and can be resolved statically.

[14]Or equivalently whose log(CBO+1) value is 0, in this case.



(a) Incoming calls (“used by”)



(b) Outgoing calls (“uses”)

Fig. 6. Breakdown of CBO results for ArgoUML

G. Dynamic class loading

Dynamic class loading is one of the reflection mechanisms supported by modern programming languages such as Java, that are rapidly gaining popularity. Table IV shows the statistics for the usage of reflective features in both ArgoUML and Azureus. The results of our investigation show that both applications make a non-trivial use of reflection, and in particular use it to load additional application classes during execution (the `forName` column). In light of this information, it is clear that proper care must be taken in order to account for such behavior when computing coupling measures. Comparing the CBO and RFC values for VTA and VTAd in Figure 4 indicates that the difference in CBO values for ArgoUML due to dynamic loading is very significant; failure to account for such features would artificially distort the distribution of CBO values and render their usage difficult. This is due to the “pluggable” architecture of ArgoUML. In ArgoUML, the entire `org.argouml.uml.cognitive.critics` package uses of this plug-in architecture. Consequently, a significant number of classes are disconnected from the call graph when we do not consider reflection.

For Azureus, we observed that the distribution of CBO calculated by VTA and VTAd are virtually indistinguishable even though the application uses reflective features in source code. Consequently, we believe that dynamic class loading does not play a major role in practice for Azureus. In fact, a manual inspection of the source code reveals that Azureus often performs dynamic loading of a predetermined class by using a fixed string constant that contains its name.

TABLE IV
USE OF DYNAMIC CLASS-LOADING

	`forName`	`newInstance`	`invoke`
ArgoUML	14	23	6
Azureus	6	7	1

H. General discussion

The results indicate that for programs that feature a non-trivial class hierarchy and a significant use of polymorphism, such as ArgoUML, the choice of call graph building algorithm used to compute coupling metrics can have an important impact on the computed values. Contrary to our expectations, the DT algorithm seems to produce CBO and RFC results that are much closer to those obtained by VTAd than we originally believed. Given the very low execution time associated with this algorithm and the low complexity of its implementation, it is not surprising to see that it is widely used in commercial and open-source tools alike. The other algorithms, however, offer sufficient precision improvements to justify their costs in certain cases.

When deciding how to implement a metric tool, one needs to consider how the metrics will be used. Tool builders and users interested in measuring coupling from a structural perspective will prefer to use the DT approach; others interested in the behavioral aspect of coupling will want to consider alternative algorithms. For example, users focusing on program understanding tasks might want an abstract view of the code to avoid inspecting all executable implementations of an interface or a class; in this case, DT might be the prefered approach. However, when using CBO in the context of change impact or error propagation, it is important to be able to resolve calls

as precisely as possible in order to avoid false negatives when estimating the impact set of a class. For such cases, basing coupling on a call graph would be important, and the different in precision between CHA and VTAd would almost always be worth the extra cost in analysis time.

An another aspect a tool developer needs to consider whether or not the whole program needs to be analysed to build a call graph. Sophisticated algorithms like VTA need an entry point from which they start and explore the dataflow through the whole program. Consequently, they need to have access to the whole code base and cannot be applied to partial programs like libraries, unlike CHA. We believe that RTA could be an interesting alternative to more sophisticated algorithms as its precision is comparable to VTA and VTAd. Furthermore, it should be possible to write an optimized implementation of RTA to achieve performance comparable to that of CHA.

V. Conclusion

Software structural metrics are powerful tools that are used in many software development and maintenance activities such as effort estimation, quality assessment, and test planning. The precision with which the structural attributes are quantified has a considerable impact on the accuracy of these activities. Coupling is among software attributes that are difficult to measure precisely. Indeed, coupling is determined in part by dynamic features that are difficult to analyze statically.

We investigated the tradeoff between cost and precision of coupling metric computation. Starting from formal definitions of some coupling metrics, we reformulated the computation algorithms in terms of call graph analysis. Then, we showed that the metric computation precision depends on the ability of the call graphs to capture accurately the dependencies between program elements. To produce the call graphs corresponding to different accuracies, we used four type-analysis algorithms, each having a computation cost. For comparison purposes, we also implemented an algorithm that extracts the coupling metrics without considering dynamic features.

To evaluate the different possibilities, we applied the computation algorithms on two large scale systems (more than 1200 classes each). One important finding is that sophisticated computation methods are necessary when capturing coupling for system where dynamic features are used. In such situations, coupling could be under- or overestimated by the classical computation methods.

Our results provide compelling evidence for the precise quantification of coupling via type-analysis. However, many open issues are still to address. The most important one is that we did not assess whether the observed variations in precision among the computation algorithms impact the accuracy of analyses that use the metrics. Future work should therefore examine whether some empirical results, established on the basis of these metrics (*e.g.,* [4]), could be challenged.

Acknowledgment

This work has been partly funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Trans. Softw. Eng.*, vol. 30, no. 8, pp. 491–506, 2004.

[2] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 1996, pp. 324–341.

[3] E. Bodden, A. Sewe, J. Sinschek, and M. Mezini, "Taming reflection: Static analysis in the presence of reflection and custom class loaders," Mar. 2010.

[4] L. C. Briand and J. Wüst, "Empirical studies of quality models in object-oriented systems," *Advances in Computers*, vol. 56, pp. 97–166, 2002.

[5] L. Briand, J. Daly, and J. Wüst, "A unified framework for coupling measurement in object-oriented systems," *IEEE Trans. on Soft. Eng.*, vol. 25, no. 1, pp. 91–121, jan/feb 1999.

[6] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Soft. Eng.*, vol. 20, no. 6, pp. 476–493, jun 1994.

[7] I. D. Coman, A. Sillitti, and G. Succi, "A case-study on using an automated in-process software engineering measurement and analysis system in an industrial environment," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 89–99.

[8] D. P. Darcy and C. F. Kemerer, "Oo metrics in practice," *IEEE Software*, vol. 22, pp. 17–19, 2005.

[9] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 1995, pp. 77–101.

[10] A. Gopal, M. S. Krishnan, T. Mukhopadhyay, and D. R. Goldenson, "Measurement programs in software development: Determinants of success," *IEEE Trans. Softw. Eng.*, vol. 28, no. 9, pp. 863–875, 2002.

[11] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 1997, pp. 108–124.

[12] M. Harman, M. Okunlawon, B. Sivagurunathan, and S. Danicic, "Slice-based measurement of coupling," in *IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution*, 1997, pp. 28–32.

[13] T. M. Meyers and D. Binkley, "An empirical study of slice-based cohesion and coupling metrics," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 1, pp. 1–27, 2007.

[14] H. Sahraoui, L. Briand, Y.-G. Guéhéneuc, and O. Beaurepaire, "Investigating the impact of a measurement program on software quality," *Information and Software Technology Journal*, 2010.

[15] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2000, pp. 264–280.

[16] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2000, pp. 281–293.

[17] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the Soot framework: Is it feasible?" in *International Conference on Compiler Construction (CC)*, 2000, pp. 18–34.

[18] S. M. Yacoub and H. H. Ammar, "A methodology for architecture-level reliability risk analysis," *IEEE Trans. Softw. Eng.*, vol. 28, no. 6, pp. 529–547, 2002.

[19] S. M. Yacoub, H. H. Ammar, and T. Robinson, "Dynamic metrics for object oriented designs," in *IEEE International Symposium on Software Metrics*, 1999, pp. 50–61.

[20] A. Zaidman and S. Demeyer, "Automatic identification of key classes in a software system using webmining techniques," *J. Softw. Maint. Evol.*, vol. 20, no. 6, pp. 387–417, 2008.