Deriving Coupling Metrics from Call Graphs

Simon Allier, Stéphane Vaucher,

Bruno Dufour, Houari Sahraoui

GEODES

DIRO,
Université de Montréal

Software metrics

- Software metrics are widely used for:
  - Quantifying software quality using models
  - Predicting software attributes (e.g. fault-proneness)
  - Summarizing complex systems
  - Studying the evolution of software systems over time
  - …
- Metrics are often defined in high-level, language-agnostic ways

Ambiguity in metric definitions

- Metric definitions use high-level concepts that leave room for different interpretations
  - e.g. "class *c* uses class *d*"
- Even attempts to formalize metric definitions usually result in ambiguity
  - e.g. "methods from class *c*"
- The same metric definition can lead to different tool implementations
- Different choices to resolve ambiguity can lead to wide variations in metric values

Example - Coupling Between Objects (CBO)

- Two distinct classes *c* and *d* are **coupled** if either
  - *c* uses *d*, or
  - *d* uses *c*
- A class *c* **uses** a class *d* if either
  - *c* calls at least one method from *d*, or
  - *c* reads or writes at least one field from *d*

Q: How to compute the set of classes used by *c* without executing the program?

How existing tools compute CBO

Tool	Considers method invocations?
Together	✓ Uses declared targets
CKJM	✓ Uses declared targets
MASU	✓ Uses declared targets
POM	✓ Uses declared targets
Aivosto	✓ Uses declared types
Jhawk	✗ Counts referenced types
Powertools	✗ Counts association types
McCabe IQ	✗ Counts external references

The tools exhibit a wide number of variations on the same definition

Goals

- Study several factors that can vary between metric implementations for a sample of existing metrics
  - In this talk, we use CBO as a running example
- Evaluate the impact of these factors on computed metric result
  - We focus on two factors: polymorphism and dynamic class loading (other factors are fixed)

Outline

- Formalization of CBO definition for dynamic language features
- Empirical study
- Related work & conclusions

A more precise definition of CBO

- Recall that two distinct classes *c* and *d* are **coupled** if either
  - *c* uses *d*, or
  - *d* uses *c*
- A class *c* **uses** a class *d* if either
  - *c polymorphically invokes* at least one method *implemented* in *d*, or
  - *c* reads or writes at least one field *implemented* in *d*

(Note: « implemented in *d* » excludes superclasses)

Polymorphically invoked methods

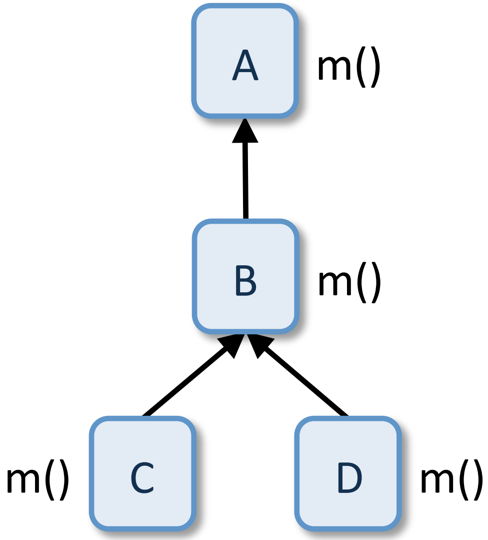
- Given a call in method *m*, how to determine the set of all methods that can be invoked at runtime?
  - This is a well-studied problem in program analysis, i.e. call graph construction
  - Several algorithms exist that make various tradeoffs between cost and precision

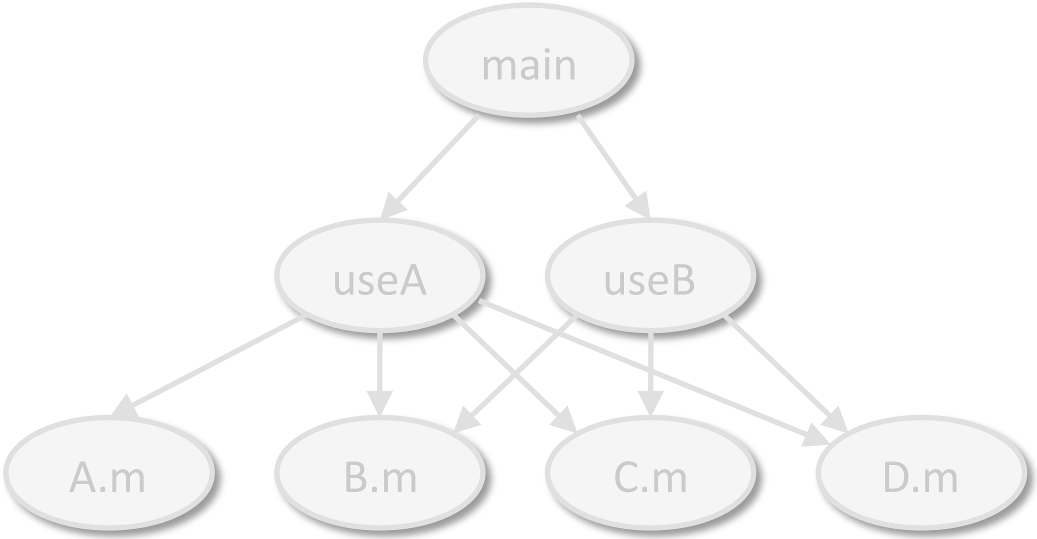
Call graph construction

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

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

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



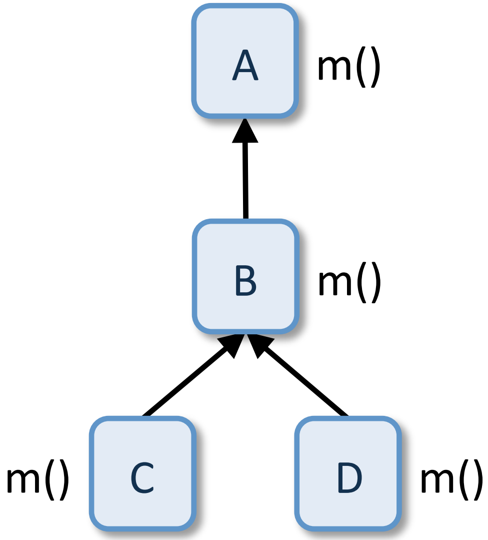


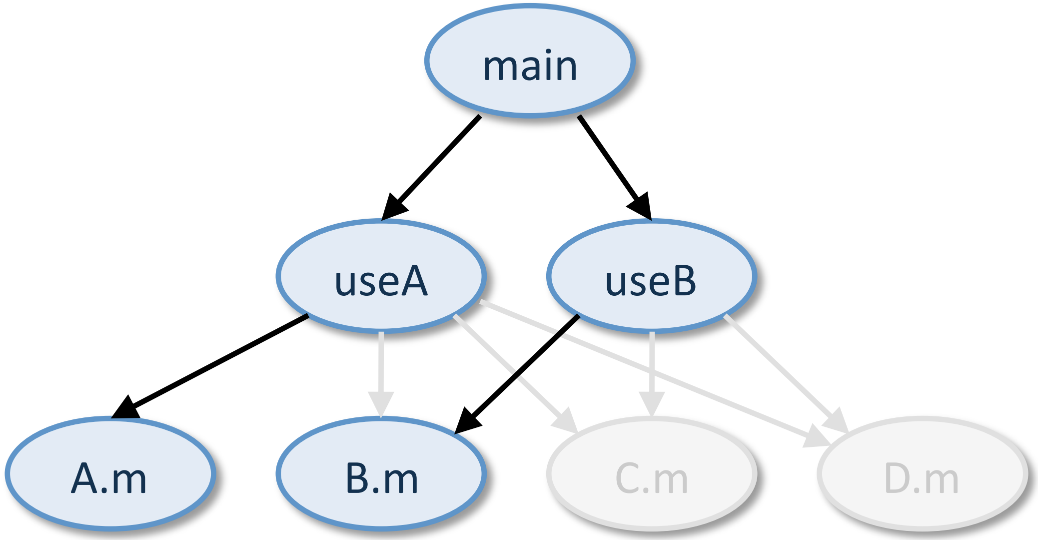
Call graph construction

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

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

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





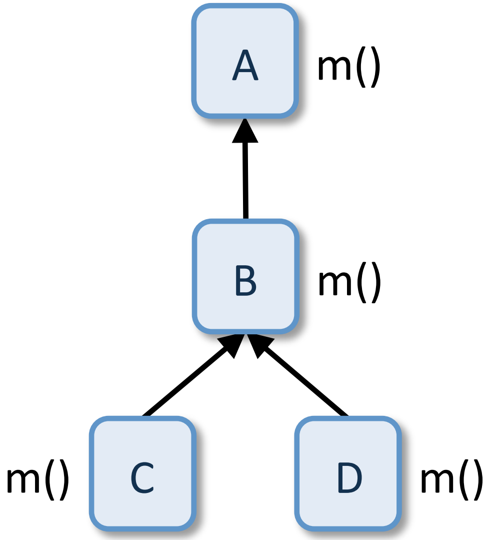
Declared Target (DT)

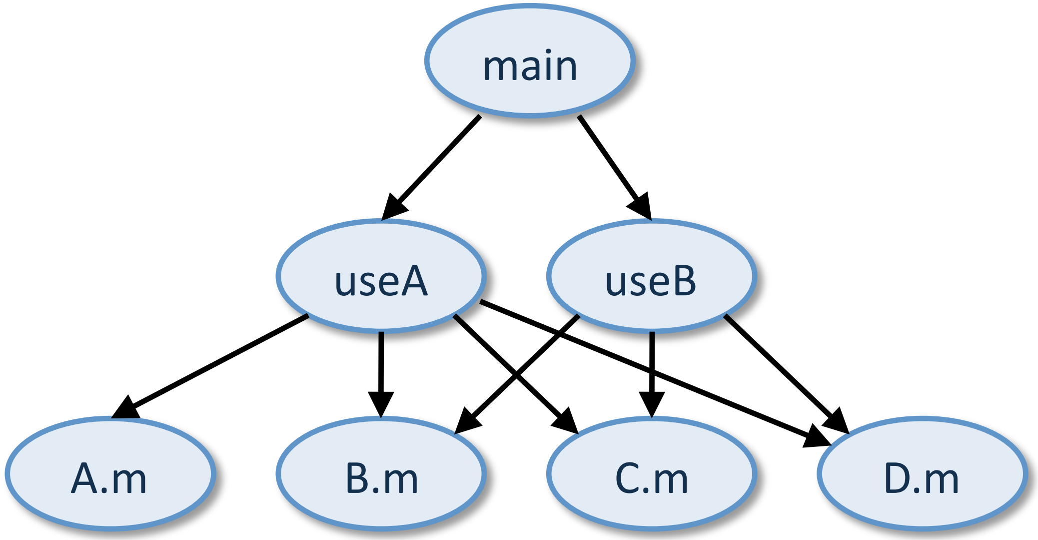
Call graph construction

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

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

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





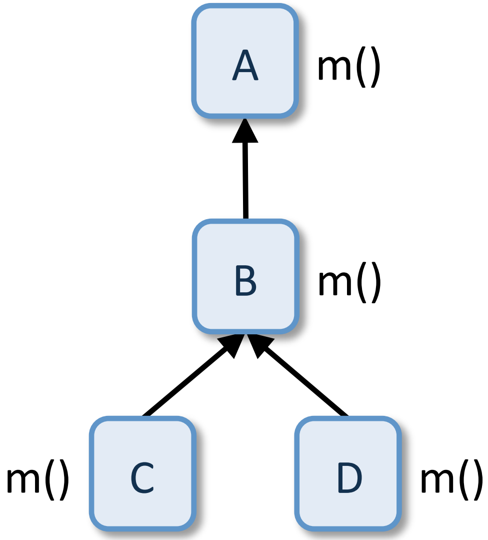
Class Hierarchy Analysis (CHA)

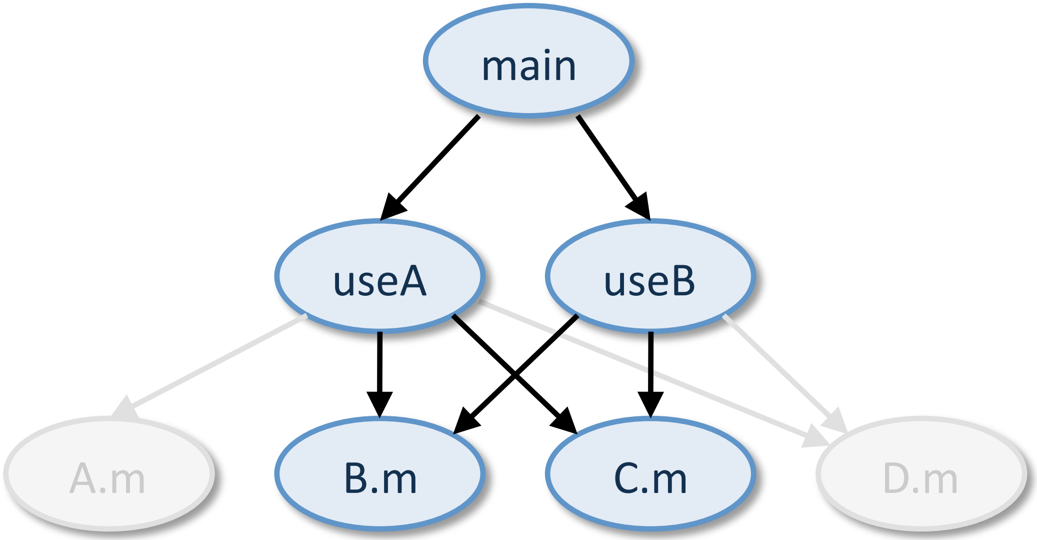
Call graph construction

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

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

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





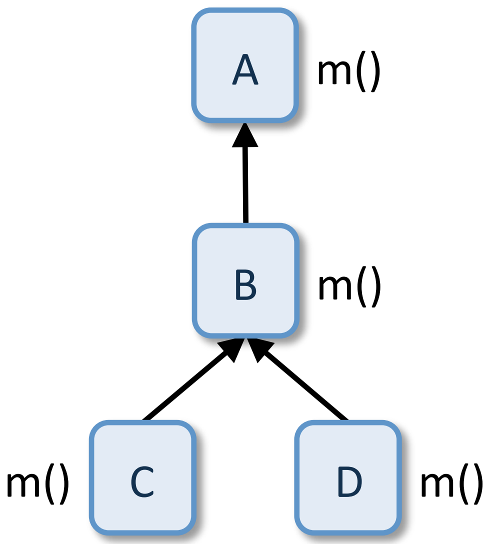
Rapid Type Analysis (RTA)

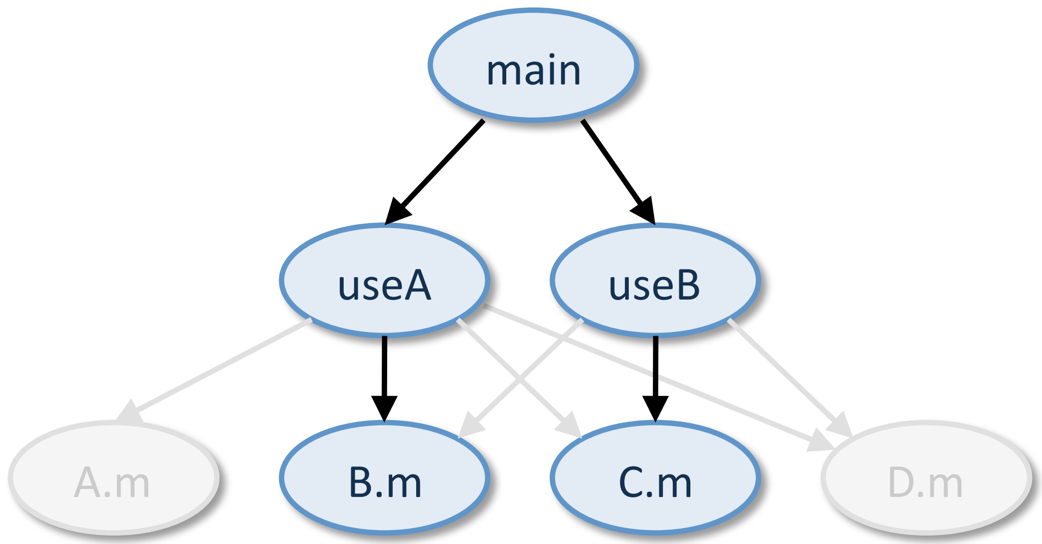
Call graph construction

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

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

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





Variable Type Analysis (VTA)

Dynamic class loading

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

- Two main strategies:
  - Ignore dynamic class loading
  - Assume all application classes can be loaded reflectively
    - To avoid imprecision, we ignore calls to no-arg constructors from `newInstance`

Experiments

Experimental setting

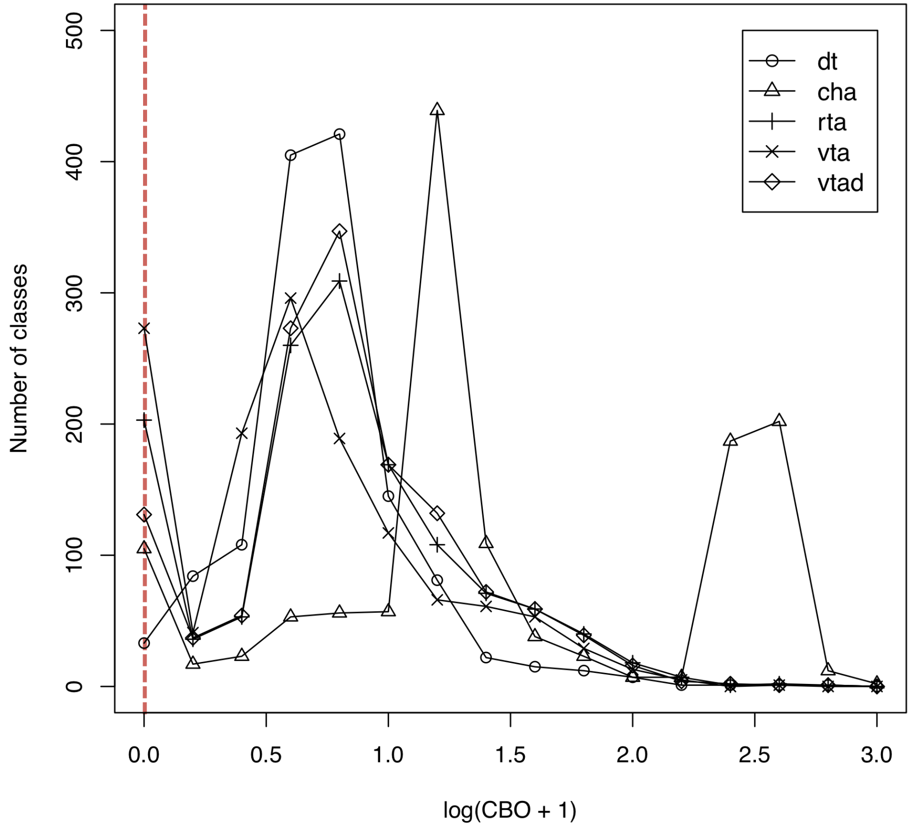
Benchmark	Classes	Interfaces
ArgoUML 0.18.1	1237	100
Azureus 2.1.0.0	1232	250

- 5 call graph algorithms implemented using Soot:
  - DT, CHA, RTA
  - VTA (no dynamic class loading)
  - VTAd (supports dynamic class loading)
- IBM JVM 6.0, Opteron 2Ghz, 8GB RAM, FC7 Linux

Call graph sizes

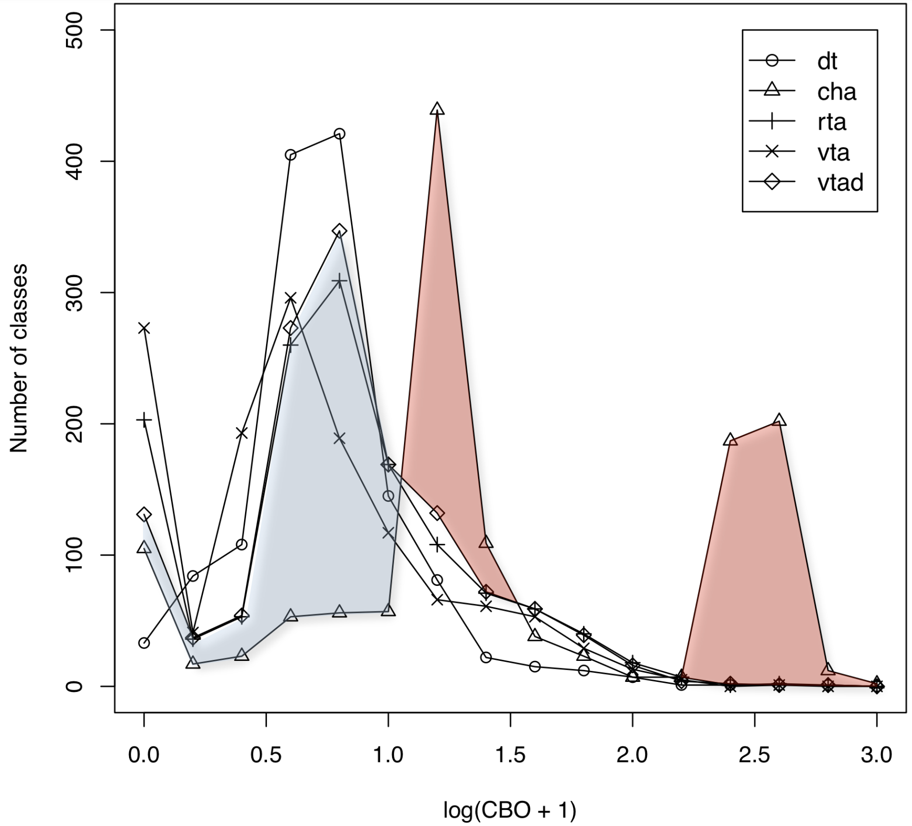
	ArgoUML		Azureus	
Algorithm	Nodes	Edges	Nodes	Edges
CHA	36 872	1 113 377	27 825	384 330
RTA	36 642	1 102 549	27 749	383 650
VTA	32 085	715 109	25 377	279 392
VTAd	36 632	1 858 348	27 076	613 025

Dead code



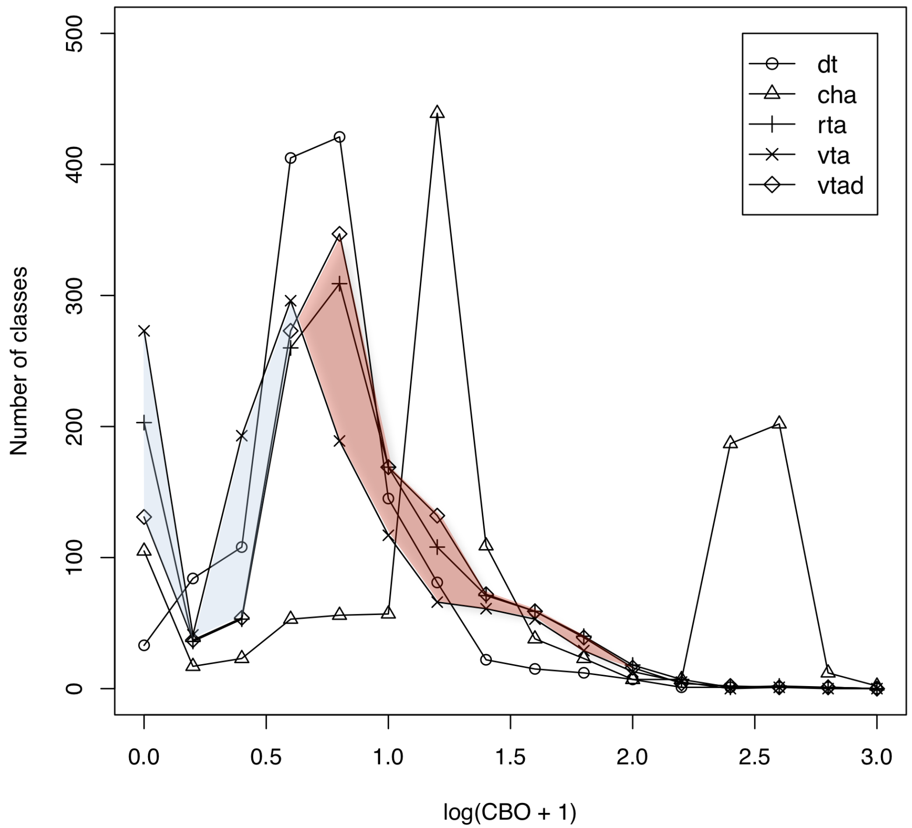
- Conservative algorithms (CHA and VTAd) can underestimate the amount of dead code
- Unsafe algorithms (DT) can both underapproximate and overapproximate the amount of dead code

Polymorphism



- DT algorithm can underapproximate the coupling as compared to VTAd for both CBO-In and CBO-Out
- CHA can mainly overapproximate CBO-In

Dynamic class loading



- Very significant difference in CBO between VTA and VTAd due to a non-trivial use of dynamic loading

Related work

- Static coupling metrics
  - e.g. Chidamber and Kemerer, Briand et al., Briand & Wüst
- Dynamic coupling metrics
  - e.g. Arisholm *et al.,* Yacoub *et al.*
- Metrics & program analysis
  - e.g. Harman *et al.,* Myers & Binkley
- Comparing software metrics tools
  - e.g. Lincke *et al.*

Conclusions

- Sophisticated computation methods are necessary when capturing coupling in the presence of dynamic features
- For programs with a non-trivial class hierarchy and a significant use of polymorphism, the choice of CG building algorithm can have an important impact on the computed coupling
- When deciding how to implement a metric tool, one needs to consider how the metrics will be used
  - e.g. program understanding vs. change impact

Additional slides

Running times

	ArgoUML			Azureus		
Algorithm	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