Deriving Coupling Metrics from Call Graphs

Simon Allier, Stéphane Vaucher, Bruno Dufour, Houari Sahraoui
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

<table>
<thead>
<tr>
<th>Tool</th>
<th>Considers method invocations?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Together</td>
<td>✓ Uses declared targets</td>
</tr>
<tr>
<td>CKJM</td>
<td>✓ Uses declared targets</td>
</tr>
<tr>
<td>MASU</td>
<td>✓ Uses declared targets</td>
</tr>
<tr>
<td>POM</td>
<td>✓ Uses declared targets</td>
</tr>
<tr>
<td>Aivosto</td>
<td>✓ Uses declared types</td>
</tr>
<tr>
<td>Jhawk</td>
<td>✗ Counts referenced types</td>
</tr>
<tr>
<td>Powertools</td>
<td>✗ Counts association types</td>
</tr>
<tr>
<td>McCabe IQ</td>
<td>✗ Counts external references</td>
</tr>
</tbody>
</table>

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)
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
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();
}
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();
}

A
    m()
    B
        m()
        C
            m()
            D
                m()

main
    useA
        A.m
    useB
        B.m
        C.m
        D.m

Declared Target (DT)
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();
}
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)
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

Variable Type Analysis (VTA)
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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Classes</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML 0.18.1</td>
<td>1237</td>
<td>100</td>
</tr>
<tr>
<td>Azureus 2.1.0.0</td>
<td>1232</td>
<td>250</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>ArgoUML</th>
<th>Azureus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>Edges</td>
</tr>
<tr>
<td>CHA</td>
<td>36 872</td>
<td>1 113 377</td>
</tr>
<tr>
<td>RTA</td>
<td>36 642</td>
<td>1 102 549</td>
</tr>
<tr>
<td>VTA</td>
<td>32 085</td>
<td>715 109</td>
</tr>
<tr>
<td>VTAd</td>
<td>36 632</td>
<td>1 858 348</td>
</tr>
</tbody>
</table>
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.
- 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
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>ArgoUML</th>
<th></th>
<th></th>
<th>Azureus</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CG</td>
<td>Metrics</td>
<td>Total</td>
<td>CG</td>
<td>Metrics</td>
<td>Total</td>
</tr>
<tr>
<td>DT</td>
<td>0:00</td>
<td>0:49</td>
<td>0:49</td>
<td>0:00</td>
<td>0:48</td>
<td>0:48</td>
</tr>
<tr>
<td>VTA</td>
<td>12:42</td>
<td>2:31</td>
<td>15:13</td>
<td>7:30</td>
<td>0:50</td>
<td>8:20</td>
</tr>
</tbody>
</table>