Introducing Rascal for meta programming

and

Eyeballing the Cyclomatic Complexity Metric

Jurgen Vinju
@RMOD, INRIA Lille

May 11th 2012
SoftWare Analysis and Transformation

Meta programming & DSLs

Parsing, Term Rewriting

ASF+SDF, Rascal, ToolBus, ATerm

Analysis and Transformation based on rELiAble tool coMpositionS

RScript, Rascal

Eclipse IMP
Credits

How Tijs & I were drafted...

- Esprit: GIPE I & GIPE II (90’s)
- ASF+SDF Meta-Environment (00’s)
- Eclipse
- IDE Meta Tooling Platform (IMP)
- Rascal is a part of IMP now
- Rascal draws inspiration from countless other projects (see SCAM 2009 paper for references)
Why?
- Why does CWI:SEN1 invest in a meta-programming language?
- Why does UvA, OU, et al. teach it?

What?
- What is it from a bird’s eye view
- What is it used for? (one example)
We study software systems: their design, their construction and their inevitable evolution.

- learning to understand software systems
- learning to improve them
- focusing on complexity as the primary quality attribute
- studying the causes of software complexity
- studying solutions to get simpler software

(NASA mission control, apollo 13)
Software is not so difficult to understand, but it is extremely complex
Software - large and complex structures of computer instructions, written and read by man, executed by computers

“marked by a senseless, disorienting, often menacing complexity…” (Infoplease.com)
The source code of "ls"

3894 lines

367 ifs

174 cases
A normal company may own $3 \times 10^{10}$ lines of code - 750,000,000 single column pages.

It goes a few times around the globe, if printed.

At 1 minute per page (?) that might take approximately 1427 years to read.

Ergo, nobody has ever understood it, or will ever fully understand it.
Why we need Rascal
@CWInl

Every week a new tool a new DSL
Rascal is a DSL for meta programming.
Rascal is a DSL for meta programming = moving between representations of source code

(Bruceghel, Tower of Babel)
Rascal is/will be a "ONE-STOP-SHOP" for analysis, transformation, generation, visualization, IDE construction, etc. (meta programming)
3 Meta Software Challenges

1: Diversity
3 Meta Software Challenges

2: Multi-disciplinary

(Raphael, Parnassus)

Saturday, May 12, 12
3 Meta Software Challenges

3: Precision vs. Efficiency
Ingredients
Ingredients

- Familiar notation
- IDE integration
- Interactive Documentation

Key enablers
Ingredients

Integration to tackle multi-disciplinary nature

Relational Calculus

Term Rewriting

Syntax definition

Familiar notation

IDE integration

Interactive Documentation

Key enablers
Ingredients

- Language parametric
- Generic programming
- Modularity

Programming techniques for dealing with diversity and scale

Integration to tackle multidisciplinary nature

Relational Calculus

Term Rewriting

Syntax definition

Familiar notation

IDE integration

Interactive Documentation

Key enablers
module FileTypes

import experiments::VL::VLCore;
import experiments::VL::Chart;
import viz::VLRender;
import JDT;
import Java;
import Resources;
import IO;
import Set;
import Map;
import Relation;
import Graph;

loc project = {project:/org.eclipse.imp.pdb.values};

Resource extract(){
    println("reading project ...");
    return extractProject(project);
}

public void main(){
    res = extract();
    extCnt = 0;
    visit(res)
    case file(loc l):
        if(l.extension != ".") extCnt[l.extension]?0 ++ = 1;
    }

    render(pieChart("File types", extCnt));}
Rascal is a domain specific programming language for software research.
Eyeballing Cyclomatic Complexity Metric

- Ongoing work with Mike Godfrey
- Typical application of Rascal
Cyclomatic Complexity

- Simple metric
- More and more popular
- Finding “complex” code
- It is a metric!
- But what does it measure?
McCabe Cyclomatic Complexity

- Is defined on the control flow graph of a procedure/method/unit of code
- Measures the number of linear independent paths through the code
- Upperbound for the number of tests at least needed
- Indicate understandability because ...
Figure 1. These two C-language snippets have the same functionality and the same CC value, yet the structured version on the right seems much simpler to understand. The procedural code is usually considered to be easier to understand — and harder to misunderstand — than its unstructured equivalent, yet the CC metric does not distinguish between them.

At the same time, there exist control flow idioms that lead to high CC, yet would seem to be fairly easy to understand. For example, a large state machine that is implemented as a number of `switch` statements with a `case` for each outgoing edge of each state will result in high CC. Yet this design pattern seems easy to grasp conceptually, since it conforms to our mental model of a state-machine, and each of the `case` statements has the same general shape: test a condition, then activate the next state. So, a high CC value may predict low understandability where the code is in fact fairly easy to understand; that is, CC may have false positives.

The case for CC in-the-large — i.e., aggregated over a large software system — is that systems that have many methods with high CC generally exhibit more bugs and higher maintenance costs [4]. For example, the SIG maintainability model aggregates CC by counting the percentage of LOC that contribute to methods with a high CC (> 10) as compared to the total LOC of a system [5]. Their model is applied on a daily basis to rapidly identify the "suspect" parts of large software systems.

Although the correlation of high aggregated CC with higher-than-expected maintenance problems has intuitive appeal, there may be several underlying factors at play (Figure 2). For example, the CC metric has been shown to correlate strongly with method size [3]. So, if a large system has many methods with high CC, then these are probably also the longer methods; in turn, this may indicate an inability of the programmers to form coherent abstractions and build robust, reusable units of code. So, is it this inability for high quality design that is causing poor understandability in many different ways, or is it just the high CC values?

To the best of our knowledge, there has been no analysis yet published that isolates the CC metric from other factors concerning software understanding and explicitly addresses the influence of CC on the effectiveness of programmers while doing maintenance. This paper aims to shed light in this area by studying the varieties of code control flow patterns across a set of large open source Java systems.

Contribution.

In this paper we investigate the relation between the shape of control flow patterns observed in Java methods to their CC metric values. We introduce the notions of abstract control flow patterns and compressed control flow patterns, which allow us to produce statistical evidence that the CC metric indeed does not adequately model the likely complexity of control flow in Java methods.

2 Observing control flow patterns

The control flow graph of a method is constructed from statements such as `if`, `while`, `break`, and `return` that may break the "straight line" flow of execution (Table 1 has a full list for Java). These statements define the shape of the control flow graph, each adding nodes and edges.

2

The CC metric makes a big conceptual leap in abstracting the shape of a method. It characterizes the control flow graph as simply the sum of the fan-outs of its nodes, and in so doing it flattens the dimensionality of the graph into a 2

Some definitions of CC model expressions, such as logical AND and OR, that can cause different branching behaviour due to short circuit evaluation. For simplicity, we consider control flow only at the statement level.
Figure 1. These two C-language snippets have the same functionality and the same CC value, yet the structured version on the right seems much simpler to understand.

At the same time, there exist control flow idioms that lead to high CC, yet would seem to be fairly easy to understand. For example, a large state machine that is implemented as a number of `switch` statements with a `case` for each outgoing edge of each state will result in high CC. Yet this design pattern seems easy to grasp conceptually, since it conforms to our mental model of a state-machine, and each of the `case` statements has the same general shape: test a condition, then activate the next state. So, a high CC value may predict low understandability where the code is in fact fairly easy to understand; that is, CC may have false positives.

The case for CC in-the-large—i.e., aggregated over a large software system—is that systems that have many methods with high CC generally exhibit more bugs and higher maintenance costs. For example, the SIG maintainability model aggregates CC by counting the percentage of LOC that contribute to methods with a high CC (＞10) as compared to the total LOC of a system. Their model is applied on a daily basis to rapidly identify the "suspect" parts of large software systems.

Although the correlation of high aggregated CC with higher-than-expected maintenance problems has intuitive appeal, there may be several underlying factors at play (Figure 2). For example, the CC metric has been shown to correlate strongly with method size. So, if a large system has many methods with high CC, then these are probably also the longer methods; in turn, this may indicate an inability of the programmers to form coherent abstractions and build robust, reusable units of code. So, is it this inability for high quality design that is causing poor understandability in many different ways, or is it just the high CC values?

To the best of our knowledge, there has been no analysis yet published that isolates the CC metric from other factors concerning software understanding and explicitly addresses the influence of CC on the effectiveness of programmers while doing maintenance. This paper aims to shed light in this area by studying the varieties of code control flow patterns across a set of large open source Java systems.

**Contributions.** In this paper we investigate the relation between the shape of control flow patterns observed in Java methods to their CC metric values. We introduce the notions of abstract control flow patterns and compressed control flow patterns, which allow us to produce statistical evidence that the CC metric indeed does not adequately model the likely complexity of control flow in Java methods.
And...

```java
switch(⊥) {
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
}
```
Questions

What is the basis of CC measuring understandability?

What does control flow look like in the first place?

Initial questions

how many times does CC make methods look more complex than they really are? (false positives)

how many times does CC make methods look simpler than they really are? (false negatives)
Problem

- There are too many methods in the world to study
- How can we generalize over something that is so utterly diverse?
Idea: control flow patterns

Use a meta programming solution! (of course!)

```plaintext
while (x >= 0) {
  if (x % 2 == 0)
    print("even");
  x--;
}
return 1;
```
```plaintext
while (⊥) {
  if (⊥)
    ⊥
}
return ⊥;
```
public bool isFork(AstNode p) =
    p is doStatement
    || p is enhancedForStatement
    || p is forStatement
    || p is ifStatement
    || p is switchCase
    || p is catchClause
    || p is whileStatement
    ;

public AstNode reduce(AstNode body) = visit(body) {
    case methodDeclaration(_,_,_,_,_,_,bl) => methodBody(bl)
    case AstNode p => noop() when ! isControl(p)
    case str x => ""
    case block([]) => noop()
    case block([AstNode n[]]) => n
    case some(noop()) => y when Option[AstNode] y := none()
    case x:[list[AstNode] a, noop(), list[AstNode] b] => y when list[AstNode] y := [e | AstNode e in b]
2.1 Computing control flow patterns

A control flow pattern (CFP) is an abstract syntax tree of a method from which all nodes that are not one of the control flow constructs have been replaced by a single keyword. Each keyword represents a type of control flow construct. For example, while, if, for, foreach, synchronized, throw, try, case, catch, do-while, block, break, continue.

Keywords that are not part of the AST of a method can easily be added to the control flow semantics, e.g., print or return. However, it is desirable to remove those nodes that are present regardless of the method body, yet because they affect the control flow of a method, can it afford to ignore important semantic details, such as fall-through and disruptive jumps.

We start by observing that the CC metric represents only a subset of all control flow statements and their net effect that removes inessential details. More precisely, we map the methods to a normalized form of program dependence graphs (PDG) as used in program slicing and token-based clone detection tools.

In order to study a very large number of methods we introduce the notion of a "control flow pattern". Instead of do not add linearly independent paths they are ignored by the CC metric.

While applying this transformation we construct a table of method representations reduced to their essential shape; this is very much in the spirit of reducing some of the detail of a control flow graph while retaining its essential shape; this is very much in the spirit of information. In this work, we seek a middle ground by introducing the notion of a "control flow pattern". Instead of do not add linearly independent paths they are ignored by the CC metric.

In theory the number of possible CFPs increases exponentially with the amount of types of control flow constructs. In practice, this is not the case, reducing patterns are found to have a very high occurrence rate; for example, the trivial patterns of "null" and "calculate-an-expression-and-return-it" had more than 2000 occurrences each.

We have collected CFPs for eight large open source Java systems:

- compendium
- Tomcat70
- dsbudget
- xml-commons-external
- apache-ant
- bcel
- hsqldb
- smallsql

Table 2 summarizes the effect of reducing methods to patterns. The CC metric is intended to measure understandability of the code. This flattening makes comparisons and diagnostics easier, but at the cost of reduced precision and loss of some important method information. In this work, we seek a middle ground by introducing the notion of a "control flow pattern". Instead of do not add linearly independent paths they are ignored by the CC metric.

Table 2. Control flow pattern statistics.

<table>
<thead>
<tr>
<th>Project</th>
<th>#Meth</th>
<th>#Pat</th>
</tr>
</thead>
<tbody>
<tr>
<td>compendium</td>
<td>7736</td>
<td>1271 (16%)</td>
</tr>
<tr>
<td>Tomcat70</td>
<td>16018</td>
<td>2211 (13%)</td>
</tr>
<tr>
<td>dsbudget</td>
<td>306</td>
<td>64 (20%)</td>
</tr>
<tr>
<td>xml-commons-external</td>
<td>3346</td>
<td>91 (2%)</td>
</tr>
<tr>
<td>apache-ant</td>
<td>10278</td>
<td>1391 (13%)</td>
</tr>
<tr>
<td>bcel</td>
<td>3076</td>
<td>286 (9%)</td>
</tr>
<tr>
<td>hsqldb</td>
<td>5326</td>
<td>1013 (19%)</td>
</tr>
<tr>
<td>smallsql</td>
<td>2556</td>
<td>353 (13%)</td>
</tr>
<tr>
<td><strong>Merged</strong></td>
<td>48642</td>
<td>5633 (11%)</td>
</tr>
</tbody>
</table>
What does CC miss?

CC metric

Patterns

Size of patterns

Measure

Patterns

Reduce

code

Measure
Cyclomatic complexity of control flow patterns (CC)

Sizes of control flow patterns (CFC)

- isoline
- 0.5 * linear fit
- linear fit
- quadratic fit

Figure 4. Comparing control flow complexity to cyclomatic complexity of control flow patterns

The CCFP for the above example would be:
```
switch(?)
{
R(case ?;)

return ?;
}
```

We can trivially extend the CFC metric to CCFPs now:

The compressed control flow complexity (CCFC) of a method is the number of nodes in the CCFP of that method.

Note that for any method $m$, $\text{CCFC}(m) \preceq \text{CFC}(m)$, but there is no such inequality for CC. The reason is that CCFC could be more than CC, as well as less.

Our hypothesis is now that there should be many methods that are highly compressible. If so, then we deduce that the CC metric overestimates control flow understandability often. Using the above definition of a CCFP we have reduced all the patterns of the systems in Table 2. This now allowed us to plot the relation between the sizes of compressed patterns and normal patterns, and their respective distribution patterns. From this we can see how often there is repetition and how much repetition there is in the control flow of real Java methods.

2.3.2 Results

In Table 2 we see that compression occurs in more than 40% of all the patterns. At the same time, the statistics show that compression does not collapse many patterns together.

In Figure 5 we can read the compression per CFP. We see that compression happens often (on all sizes of patterns), and that compression rates can be high for all sizes. Smaller patterns, if they compress, more often compress a little than a lot. Larger patterns, of which there are a lot less, do compress extremely in many cases. We have plotted linear, quadratic and square root fits (using least squares) such that it is clearly visible that the data set clearly favors small compression rates (many dots are printed on top of each other). The square root fits best for this data set, confirming that compression is more effective on larger patterns.

We have looked up a number of the larger methods to see what code would compress. In the systems we found...
The CCFP for the above example would be:

```
switch (?
{
R (case ?:
return ?;)
}
```

We can trivially extend the CFC metric to CCFPs now:

The compressed control flow complexity (CCFC) of a method is the number of nodes in the CCFP of that method.

Note that for any method $m$, $\text{CCFC}(m) \leq \text{CFC}(m)$, but there is no such inequality for CC. The reason is that CCFC could be more than CC, as well as less.

Our hypothesis is now that there should be many methods that are highly compressible. If so, then we deduce that the CC metric overestimates control flow understandability often. Using the above definition of a CCFP we have reduced all the patterns of the systems in Table 2. This now allowed us to plot the relation between the sizes of compressed patterns and normal patterns, and their respective distribution patterns. From this we can see how often there is repetition and how much repetition there is in the control flow of real Java methods.

2.3.2 Results

In Table 2 we see that compression occurs in more than 40% of all the patterns. At the same time, the statistics show that compression does not collapse many patterns together.

In Figure 5 we can read the compression per CFP. We see that compression happens often (on all sizes of patterns), and that compression rates can be high for all sizes. Smaller patterns, if they compress, more often compress a little than a lot. Larger patterns, of which there are a lot less, do compress extremely in many cases. We have plotted linear, quadratic and square root fits (using least squares) such that it is clearly visible that the data set clearly favors small compression rates (many dots are printed on top of each other). The square root fits best for this data set, confirming that compression is more effective on larger patterns.

We have looked up a number of the larger methods to see what code would compress. In the systems we found 6
The CCFP for the above example would be:

```java
switch (?
{
R (case ?:
return ?;
}
```

We can trivially extend the CFC metric to CCFPs now:

The compressed control flow complexity (CCFC) of a method is the number of nodes in the CCFP of that method.

Note that for any method $m$, $\text{CCFC}(m) \leq \text{CFC}(m)$, but there is no such inequality for $\text{CC}$. The reason is that CCFC could be more than CC, as well as less.

Our hypothesis is now that there should be many methods that are highly compressible. If so, then we deduce that the CC metric overestimates control flow understandability often. Using the above definition of a CCFP we have reduced all the patterns of the systems in Table 2. This now allowed us to plot the relation between the sizes of compressed patterns and normal patterns, and their respective distribution patterns. From this we can see how often there is repetition and how much repetition there is in the control flow of real Java methods.

### 2.3.2 Results

In Table 2 we see that compression occurs in more than 40% of all the patterns. At the same time, the statistics show that compression does not collapse many patterns together.

In Figure 5 we can read the compression per CFP. We see that compression happens often (on all sizes of patterns), and that compression rates can be high for all sizes. Smaller patterns, if they compress, more often compress a little than a lot. Larger patterns, of which there are a lot less, do compress extremely in many cases.

We have plotted linear, quadratic and square root fits (using least squares) such that it is clearly visible that the data set clearly favors small compression rates (many dots are printed on top of each other). The square root fits best for this data set, confirming that compression is more effective on larger patterns.

We have looked up a number of the larger methods to see what code would compress. In the systems we found...
So

- If we assume that to understand a method you have to understand all of its control flow
- Then CC in theory does not measure accurately enough
- And in practise CC indeed does not predict the sizes of methods at all
- So we have to do more work
Next idea: compress!

```
switch(⊥) {
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;  switch (⊥) {
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;  R (case ⊥ : return ⊥;)
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
    case ⊥ : return ⊥;
}
```
public AstNode compress(AstNode body) = innermost visit(body) {
  case [*a, repeated([*n]), n, *b] => [*a, repeated([*n]), *b]
  case [*a, x, *c, x, c, *d] => [*a, repeated([x,*c]), *d]
  case block(repeated(n)) => repeated(n)
}
the most extreme compression would be found in generated code (by lexer and parser generators). This is not surprising, but it is interesting to see that such generated patterns are compressible by simply eliminating repetition. The following is an example of a nested compressed CFP, found in the smallsql system. 

```
switch (?
{{
 R (case ?:
 { return ?; })
}})
```

We simplified it here for presentation purposes, by leaving out the context around the switch and removing some irregular cases. The pattern was associated to a single method which interprets boolean expressions in SQL with a CC of 126. The CCFC of this method is 75, and the CC of the compressed pattern is 27 (as opposed to the shown pattern which has CCFC equal to 7 and CC equal to 3). The method dispatches on the types of the arguments of an expression with the outer switch, and then on the operator kind in the nested switch and computes the result of the expression via recursion. This is a simple design for an interpreter that is trivial to understand. In theory, the particular nested pattern may represent a quadratic amount of methods, depending on how many repetitions occur at each level. In practice, we found that it occurs only once in the systems that we investigated.

The method with the largest compressed size (CCFC) was found in compendium: 179. It has an original CC of 141, its CFC is 198 and the CC of the compressed pattern is still 119. This is the worst case compression rate we found for such larger methods. The code dispatches on key press events and directly implements the associated actions. We can conclude that compression may be used to filter large methods that are easy-to-understand patterns and perhaps even generated. However, there are so few of such larger patterns that we should not jump to the conclusion that CC is not a good way of finding hard-to-understand patterns. For smaller patterns the compression may be less visible, yet is has significant effect on the interpretation of the metrics. Although smaller patterns are usually not compressed below 50%, the compression does affect the interpretation of the metrics via the commonly used threshold of 10. From this perspective we can learn that systems that are easy to understand because they have repetitive control flow structure.
Figure 6. Size and CC distributions with and without compression

We conclude that CC indeed often underestimates the understandability of CFPs; it is most pronounced in larger methods (which are much less common), but it is still significant in the shorter methods (which are very common). This suggests that the CC metric is not very accurate for judging individual methods on understandability, and that when used for aggregation over whole systems using a threshold care must be taken in interpreting the results if there are many methods whose CFC are in the range of 10 to 20.

3 Related work

There is a large body of work on the generation, interpretation, and experimental validation of software metrics (i.e., the work by Halstead [9]). We do not have the space here to enumerate it, so we mention only relevant and more recent developments.

Herraiz and Hassan argued we do not need complexity metrics because they correlate very strongly with the number of lines of code (LOC) [3]. While our results also found such a correlation, we draw a slightly different conclusion. The McCabe cyclomatic complexity metric correlates in general with the size of a method because every method has at least a few branches scattered over its body. However, this does not accurately predict the complexity of the rest of the code that may or may not use more than this minimal number of branches. We distinguish explicitly between interpretation on a method-by-method basis versus a global system-to-system aggregated comparison. Herraiz and Hassan's conclusion remains valid for the latter perspective, but on the smaller scale we feel that it is reasonable to assert that there is still room for better complexity metrics.

Vasilescu et al. [10, 11] have studied the effect of different aggregations for software metrics on their interpretation, which is very noticeable. The SIG maintainability model also pays attention directly to the effect of aggregation on their judgement of quality [5]. We have learned this lesson and steered away in the current paper from computing aggregates such as averages. Indeed, the statistical distributions shown in this paper are interpreted directly rather than framing them in a statistical model.

Jbara et al. have asked the same questions we did, and studied the Linux kernel source code to answer them. They founded evidence of code that was labelled as complex by CC, yet seemed “well structured” enough for understanding and indeed was maintained actively. Our results corroborate theirs in that respect. Jbara also concludes, like McCabe mentioned in the original paper on cyclomatic complexity [2], perhaps large switch cases should not be counted. We have taken this idea one step further and eliminated all locally repetitive structures. Our paper also adds another perspective, namely that cyclomatic complexity may miss the opportunity of spotting hard-to-understand code next to mislabeling easy code as complex.

Alves et al. studies the construction of benchmarks from software metrics [12]. They also notice that software metrics like CC are often distributed according to (what seems to be) a power law. They automatically derive threshold values in an objective and repeatable way. It may very well be that by replacing CC with CCFM some systems fall in entirely different categories in the benchmarks they produce. Our work therefore is highly relevant to the industry of software quality verification and monitoring.

Clauset et al. point out how hard it is to verify that a data set is accurately modelled by a power law distribution [13].

Saturday, May 12, 12
So

- Compression affects all method sizes
- Compression makes methods drop under 10
- Compression affects larger methods most
- Compression separates generated/simple code from the really hideous parts
We have found indications that:

- CC is not good because it misses complexity
- CC is not good because it sees complexity where there is none
- Now what? We’ll see...
Rascal is a “ONE-STOP-SHOP” for meta programming

http://www.rascal-mpl.org

Control flow patterns are a way of studying control flow

http://homepages.cwi.nl/~jurgenv