DepDigger: A Tool for Detecting Complex Low-Level Dependencies *

Dirk Beyer
Simon Fraser University, B.C., Canada
& University of Passau, Germany

Ashgan Fararooy
Simon Fraser University, B.C., Canada

Abstract

We present a tool that identifies complex data-flow dependencies on code-level, based on the measure dep-degree. Low-level dependencies between program operations are modeled by the use-def graph, which is generated from reaching definitions of variables. The tool annotates program operations with their dep-degree values, such that 'difficult' program operations are easy to locate. We hope that this tool helps detecting and preventing code degeneration, which is often a challenge in today's software projects, due to the high refactoring and restructuring frequency.

1. Tool Overview

We present a lightweight tool that helps answering the following question: Given two versions of a software program that have the same behavior and differ only in code structure, which version is to prefer in terms of low-level dependency structure? For example, if the second version is the result of changing the first version, we would like to know whether the change actually improves the code structure, i.e., was a positive 'refactoring'.

The tool is based on *dep-degree*, an indicator for complex low-level dependencies that uses the notion of reaching definitions, a well-known concept from compiler optimization and program analysis. The dep-degree for a single operation is the total number of reaching definitions for the variables that occur in the operation. The dep-degree for a set of operations (or a complete function) is the sum over all dep-degrees for operations (or the number of edges in the use-def graph, respectively). For example, consider the assignment operation $\texttt{x} = \texttt{a} - \texttt{b}$. The dep-degree for this operation is the number of different reaching definitions for variable `a` plus the number of different reaching definitions for variable `b`.

The measure dep-degree is easy to understand, simple to compute, flexible and scalable in its application, and independently complementing other indicators. In comparison to the indicators LOC and cyclomatic complexity, which are often used for measuring maintainability and understandability, dep-degree is a relatively better indicator for assessing code improvements (e.g., by refactoring). The relation of the measure dep-degree to other existing indicators is discussed in the concept paper that introduced dep-degree [1].

Design and Implementation. We have developed a software tool to automate the calculation of the dep-degree values. The tool integrates with the Eclipse IDE as plug-in, and is capable of interacting with the Eclipse source editor. The plug-in uses the Eclipse Java development tools (JDT) to parse the selected source file within the active editor, and to obtain the abstract syntax tree. Then, the control-flow graph (CFG) of the program is extracted by traversing the syntax tree using the available visitor class from the JDT. The set of reaching definitions is computed for each node in the CFG and the dep-degree value is calculated for all program operations accordingly (cf. Fig. 1).

Availability. The tool is released under the Apache 2 license and the source code as well as binaries are freely available at `http://www.sosy-lab.org/~dbeyer/DepDigger`.

Value Annotation. The tool highlights the operations of the program within the editor, based on their dep-degree value. This is done using a generated relative color map, which assigns to each dep-degree value a color on the scale from white to red, where 0 is mapped to white and the maximal dep-degree value for a program operation is mapped to red (cf. Fig. 3). It is also possible to view the exact dep-degree value for each operation in a dedicated text field or marker.

Tracking Dependencies. If an operation is selected (or clicked on) in the editor, then all defining operations for the selected operation are highlighted in cyan (Fig. 2).

Text Output. The tool also generates the results in plain text format, which contains the dep-degree value for each method in the class and the value for each operation within every method, along with the set of reaching definitions for each operation. This output can be useful for off-line or automatic post-processing.

*This paper supplements our ICPC'10 technical paper [1].

978-0-7695-4113-6/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPC.2010.52

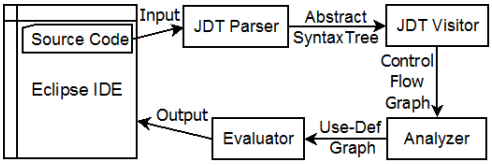

Figure 1. Tool architecture

```
4  public void test() {
5    int b = 0, c = 0;
6    if (b + c > 0)
7      c++;
8    else
9      b += 2;
10   c = 2 * c;
11 }
```

Figure 2. The defining operations (in cyan)

```
19
20        switch (obj) {
21            case 1 : result += "Pen";
22            break;
23            case 2 : result += "Pencil";
24            break;
25        }
26
27        if (obj < 0)
28            result = "Unknown Request.";
29
30        return result;
31    }
```

Figure 3. Coloring the operations

Method	Version	LOC	CC	DD
printOwing	Original	19	5	27
	Refactored	1	1	1
getOutstanding	Original	0	0	0
	Refactored	7	2	8
getTaxRate	Original	0	0	0
	Refactored	7	4	1
printDetails	Original	0	0	0
	Refactored	5	1	6
log	Original	0	0	0
	Refactored	1	1	2
TOTAL	Original	19	5	27
	Refactored	21	9	18

Table 1. Indicator values for the 'Extract Method' example before and after refactoring

2. Applications

Assessment before/after Refactoring. Dep-degree can be used to indicate structural improvements. To illustrate this, we consider a well-known refactoring example of which we know already that it is a good refactoring ('authoritative' example) and test if the indicator dep-degree agrees. There is no other simple indicator for structural improvement available, and therefore, we compare dep-degree with the widely used indicators lines of code (LOC) and cyclomatic complexity (CC). LOC measures the length of code; CC measures the difference of control-flow nodes and edges.

The 'extract method' refactoring rule recommends, for a given code fragment, to *factor out* a cohesive, common, possibly repeating 'chunk' of code and move it to a new method. We revisit (an extended version of) the method mentioned in Martin Fowler's refactoring book which prints the amount of money a customer owes (`printOwing`). We extract four new methods: 'getOutstanding', 'getTaxRate', 'printDetails', and 'log'.

Table 1 presents the indicator values for lines of code (LOC), cyclomatic complexity (CC), and dep-degree (DD). The value 0 indicates that the method was empty before the refactoring, i.e., did not exist. Only the new indicator dep-degree correctly identifies the improvement of the code (the value for `printOwing` was 27 before, and the sum over all new methods is 18); the other two indicators suggest that the new code is longer (LOC increased by two from 19 to 21) or has more complicated control flow (CC increased by four from 5 to 9).

We performed similar analyses on other examples of refactoring, including 'Parameterize Method', 'Replace Conditional with Polymorphism', and 'Pull Up Method'. Our experiments indicate that dep-degree acknowledges the effect of simplifying refactorings as code improvements.

Localization of Problematic Code. In the last paragraphs we applied our tool to the assessment of code changes as they occur during refactorings, and illustrated that the dep-degree values match the developer opinion, and that LOC and cyclomatic complexity are not applicable to assessing refactorings. Next we point out another possible application for using dep-degree: detecting problematic code, i.e., to *indicate* and *locate* pieces of code with complex dependencies (which could be considered for refactoring).

Dep-degree for program operations can be applied to single program operations. The tool generates a detailed report of the dep-degree values for each operation in the program. This immediately proposes two possible uses: (a) we can list and inspect the operations with highest dep-degree values, and (b) we can color (highlight) each operation in the source-code editor according to its dep-degree value.

Both features (a) and (b) are implemented in the Eclipse plug-in. For (a), the plug-in attaches markers to the left vertical ruler of Eclipse's source editor. These markers indicate the operations with the highest dep-degree values in the selected source file. (A parameter can be set to limit the amount of markers to values that exceed a certain threshold.) For (b), we generate a relative color map *rgb* that assigns to each dep-degree value a color on the scale from white to red (illustrated in Fig. 3).

References

[1] D. Beyer and A. Fararooy. A simple and effective measure for complex low-level dependencies. In *Proc. ICPC*. IEEE, 2010.