The Journal of Systems and Software 88 (2014) 1–24

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Coherent clusters in source code☆

Syed Islam[a,*], Jens Krinke[a], David Binkley[b], Mark Harman[a]

[a] *University College London, United Kingdom*
[b] *Loyola University Maryland, United States*

ARTICLE INFO

Article history:
Received 19 November 2012
Received in revised form 16 July 2013
Accepted 18 July 2013
Available online 21 August 2013

Keywords:
Dependence analysis
Program comprehension
Software clustering

ABSTRACT

This paper presents the results of a large scale empirical study of *coherent dependence clusters*. All statements in a coherent dependence cluster depend upon the same set of statements and affect the same set of statements; a coherent cluster's statements have 'coherent' shared backward and forward dependence. We introduce an approximation to efficiently locate coherent clusters and show that it has a *minimum* precision of 97.76%. Our empirical study also finds that, despite their tight coherence constraints, coherent dependence clusters are in abundance: 23 of the 30 programs studied have coherent clusters that contain at least 10% of the whole program. Studying patterns of clustering in these programs reveals that most programs contain multiple substantial coherent clusters. A series of subsequent case studies uncover that all clusters of significant size map to a logical functionality and correspond to a program structure. For example, we show that for the program acct, the top five coherent clusters all map to specific, yet otherwise non-obvious, functionality. Cluster visualization also brings out subtle deficiencies in program structure and identifies potential refactoring candidates. A study of inter-cluster dependence is used to highlight how coherent clusters are connected to each other, revealing higher-level structures, which can be used in reverse engineering. Finally, studies are presented to illustrate how clusters are not correlated with program faults as they remain stable during most system evolution.

1. Introduction

Program dependence analysis is a foundation for many activities in software engineering such as testing, comprehension, and impact analysis (Binkley, 2007). For example, it is essential to understand the relationships between different parts of a system when making changes and the impacts of these changes (Gallagher and Lyle, 1991). This has led to both static (Yau and Collofello, 1985; Black, 2001) and blended (static and dynamic) (Ren et al., 2006, 2005) dependence analyses of the relationships between dependence and impact.

One important property of dependence is the way in which it may cluster. This occurs when a set of statements all depend upon one another, forming a dependence cluster. Within such a cluster, any change to an element potentially affects every other element of the cluster. If such a dependence cluster is very large, then this mutual dependence clearly has implications related to the cost of maintaining the code.

In previous work (Binkley and Harman, 2005), we introduced the study of dependence clusters in terms of program slicing and demonstrated that large dependence clusters were (perhaps surprisingly) common, both in production (closed source) code and in open source code (Harman et al., 2009). Our findings over a large corpus of C code was that 89% of the programs studied contained at least one dependence cluster composed of 10% or more of the program's statements. The average size of the programs studied was 20KLoC, so these clusters of more than 10% denoted significant portions of code. We also found evidence of super-large clusters: 40% of the programs had a dependence cluster that consumed over half of the program.

More recently, our finding that large clusters are widespread in C systems has been replicated for other languages and systems by other authors, both in open source and in proprietary code (Acharya and Robinson, 2011; Beszédes et al., 2007; Szegedi et al., 2007). Large dependence clusters were also found in Java systems (Beszédes et al., 2007; Savernik, 2007; Szegedi et al., 2007) and in legacy Cobol systems (Hajnal and Forgács, 2011).

There has been interesting work on the relationship between faults, program size, and dependence clusters (Black et al., 2006), and between impact analysis and dependence clusters (Acharya and Robinson, 2011; Harman et al., 2009). Large dependence clusters can be thought of as dependence 'anti-patterns' because of the high impact that a change anywhere in the cluster has. For example, it may lead to problems for on-going software maintenance and evolution (Acharya and Robinson, 2011; Binkley et al., 2008; Savernik, 2007). As a result, refactoring has been proposed

☆ This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.
* Corresponding author.
E-mail address: s.islam@cs.ucl.ac.uk (S. Islam).

0164-1212/$ – see front matter © 2013 The Authors. Published by Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.07.040

as a technique for breaking larger clusters of dependence into smaller clusters (Binkley and Harman, 2005; Black et al., 2009).

Dependence cluster analysis is complicated by the fact that inter-procedural program dependence is non-transitive, which means that the statements in a traditional dependence cluster, though they all depend on each other, may not each depend on the same set of statements, nor need they necessarily affect the same set of statements external to the cluster.

This paper introduces and empirically studies[1] *coherent dependence clusters*. In a coherent dependence cluster all statements share identical intra-cluster and extra-cluster dependence. A coherent dependence cluster is thus more constrained than a general dependence cluster. A coherent dependence cluster retains the essential property that all statements within the cluster are mutually dependent, but adds the constraint that all incoming dependence must be identical and all outgoing dependence must also be identical. That is, all statements within a coherent cluster depend upon the same set of statements outside the cluster and all statements within a coherent cluster affect the same set of statements outside the cluster.

This means that, when studying a coherent cluster, we need to understand only a single external dependence context in order to understand the behavior of the entire cluster. For a dependence cluster that fails to meet the external constraint, statements of the cluster may have a different external dependence context. This is possible because inter-procedural dependence is non-transitive.

It might be thought that very few sets of statements would meet these additional coherence constraints, or that, where such sets of statements do meet the constraints, there would be relatively few statements in the coherent cluster so-formed. Our empirical findings provide evidence that this is not the case: coherent dependence clusters are common and they can be very large.

This paper is part of a series of work that we have conducted in the area of dependence clusters. The overarching motivation for this work is to gain a better understanding of the dependence clusters found in programs. Although this paper is a continuation of our previous work on dependence clusters, we present the work in a completely new light. In this paper we show that the specialized version of dependence clusters, *coherent* clusters are found in abundance in programs and need not be regarded as problems. We rather show that these clusters map to logical program structures which will aid developers in program comprehension and understanding. Furthermore, this paper extends the current knowledge in the area and motivates future work by presenting initial results of inter-cluster dependence which can be used as a foundation for reverse engineering. We answer several representative open questions such as whether clusters are related to program faults and how clusters change over time during system evolution.

The primary contributions of the paper are as follows:

1 An Empirical analysis of thirty programs assesses the frequency and size of coherent dependence clusters. The results demonstrate that large coherent clusters are common, validating their further study.
2 Two further empirical validation studies consider the impact of data-flow analysis precision and the precision of the approximation used to efficiently identify coherent clusters.
3 A series of four case studies shows how coherent clusters map to logical program structures.
4 A study of inter-cluster dependence highlights how coherent clusters form the building blocks of larger dependence structures where identification can support, as an example, reverse engineering.
5 A study of bug fixes finds no relationship between program faults and coherent clusters implying that dependence clusters are not responsible for program faults.
6 A longitudinal study of system evolution shows that coherent clusters remain stable during evolution thus depicting the core architecture of systems.

The remainder of this paper is organized as follows: Section 2 provides background on coherent clusters and their visualization. Section 3 provides details on the subject programs, the validation of the slice approximation used, and the experimental setup. This is followed by quantitative and qualitative studies into the existence and impact of coherent dependence clusters and the inter-cluster dependence study. It also includes studies on program faults and system evolution and their relationship to coherent clusters. Section 4 considers related work and finally, Section 5 summarizes the work presented.

2. Background

This section provides background on dependence clusters. It first presents a sequence of definitions that culminate in the definition for a coherent dependence cluster. Previous work (Binkley and Harman, 2005; Harman et al., 2009) has used the term *dependence cluster* for a particular kind of cluster, termed a *mutually-dependent cluster* herein to emphasize that such clusters consider only mutual dependence internal to the cluster. This distinction allows the definition to be extended to incorporate external dependence. The section also reviews the current graph-based visualizations for dependence clusters.

2.1. Dependence clusters

Informally, *mutually-dependent clusters* are maximal sets of program statements that mutually depend upon one another (Harman et al., 2009). They are formalized in terms of mutually dependent sets in the following definition.

Definition 2.1 *(Mutually-dependent set and cluster (Harman et al., 2009)).* A *mutually-dependent set (MDS)* is a set of statements, S, such that

$\forall x, y \in S : x$ depends on y.

A *mutually-dependent cluster* is a maximal MDS; thus, it is an MDS not properly contained within another MDS.

The definition of an MDS is parameterized by an underlying *depends-on* relation. Ideally, such a relation would precisely capture the impact, influence, and dependence between statements. Unfortunately, such a relation is not computable (Weiser, 1984). A well known approximation is based on Weiser's *program slice* (Weiser, 1984): a slice is the set of program statements that affect the values computed at a particular statement of interest (referred to as a slicing criterion). While its computation is undecidable, a minimal (or precise) slice includes exactly those program elements that affect the criterion and thus can be used to define an MDS in which t depends on s iff s is in the minimal slice taken with respect to slicing criterion t.

The slice-based definition is useful because algorithms to compute approximations to minimal slices can be used to define and compute approximations to mutually-dependent clusters. One such algorithm computes a slice as the solution to a reachability problem over a program's *System Dependence Graph* (SDG) (Horwitz et al., 1990). An SDG is comprised of vertices, which essentially represent the statements of the program and two kinds of edges: data dependence edges and control dependence edges. A data dependence connects a definition of a variable with each use of the variable

[1] Preliminary results were presented at PASTE (Islam et al., 2010b).