Dependence Anti Patterns

David Binkley†1 **Nicolas Gold**† **Mark Harman**†
Zheng Li† **Kiarash Mahdavi**† **Joachim Wegener**‡
†CREST, King's College London
Department of Computer Science
Strand, London WC2R 2LS, United Kingdom
‡Daimler AG
Alt-Moabit 96a
D-10559 Berlin

Abstract

A Dependence Anti Pattern is a dependence structure that may indicate potential problems for on–going software maintenance and evolution. Dependence anti patterns are not structures that must always be avoided. Rather, they denote warnings that should be investigated. This paper defines a set of dependence anti patterns and presents a series of case studies that show how these patterns can be identified using techniques for dependence analysis and visualization. The paper reports the results of this analysis on six real world programs, two of which are open source and four of which are part of production code in use with Daimler.

1 Introduction

The dependence structure of a program or system can be used to reveal structural properties that may have important implications for software maintenance and evolution. Because the analysis concerns the semantic properties of the system, the semantic information obtained by dependence analysis can be deep and may yield insight into potential problems for on–going evolution.

This paper introduces the concept of a 'Dependence Anti Pattern.' A dependence anti pattern is a dependence structure that may cause potential problems. Typically these problems will take the form of difficulties in comprehension, testing, reverse engineering, re-use, and maintenance. It is not the purpose of the paper to explore the impact of the presence of dependence anti patterns on these activities. Rather, the paper takes as a starting point, the belief that dependence structure has some bearing on all these activities and sets out to define several forms of dependence anti pattern and introduce techniques for locating them.

The paper takes as case study material two open source programs and four production programs all of which are in use at Daimler AG Berlin. Like many organizations, Daimler is increasingly outsourcing the development of software systems for their products. This raises the issue of quality assurance for the code that emerges from the third parties with which Daimler works and upon which the organisation relies.

Daimler's business position is that of an organisation considering taking on the considerable burden of managing the evolution of these third party programs. Therefore, Daimler seeks techniques for identifying possible problems that could be statically discovered at the code level, independent of domain knowledge, thereby posing an interesting research challenge. While there are many techniques for software quality measurement based on *process*, such as the quality maturity model of the SEI [15], there is little agreement on what constitutes a good indicator of software *product* quality.

Many metrics for software measurement have been proposed, but these are controversial when applied to assessment of software quality [7, 18, 19]. Often, these code-based metrics embody a purely syntactic assessment of software systems such as counts of syntactic features: lines of code, number of branches, depth of inheritance, number of children [5, 11, 14]. Many originally proposed syntactic metrics have been heavily criticized for, *inter alia*, being overly simplistic, lacking in semantic depth and, above all, failing to capture properties which have any correlation to software quality [6, 8, 17].

The experience of Daimler, when applying count–based code–level metrics such as these was that they failed to help differentiate poor quality from good quality third party code; thus, they could not be used to indicate future pitfalls and problems. One issue raised, which is the motivation for this work, is the lack of semantic information in the metrics. These metrics also bring the problem of determining a 'bounding value' or 'threshold' beyond which concern for future evolution costs was appropriate.

Following the Evol'2008 conference theme of bridging boundaries between academia and industry, theory and practice, and intangible and tangible, this paper seeks to combine these three strands. The paper reports the results of a research project commissioned by Daimler Berlin from the CREST centre at King's College London Department

[1] On sabbatical leave from Loyola College in Maryland.

of Computer Science. The project aims to apply theoretical techniques for dependence analysis, in order to identify potential practical problems with software in use at Daimler. The approach uses visualization to make tangible, the intangible dependence structure of the third party programs that Daimler is considering adopting. These techniques also seek to avoid the problems of the arbitrary selection of a 'metric threshold'.

The paper identifies a set of signatures in the dependence structure that can be considered potentially problematic and thus worthy of further investigation. The hope is that the semantic nature of the dependence analysis will allow the identification of deeper semantic properties than can be achieved using metrics defined purely on syntactic constructions.

Two dependence analyses are used: analysis of dependence clusters and analysis of predicate dependence. Dependence clusters contain sets of mutually inter-dependent statements. Where a program contains a large dependence cluster, software modification may cause significant ripple effects and, as a result, problems for maintainers. The size of a dependence cluster can be thus tied to the maintainability of a program at a coarse level of granularity. By contrast, predicate dependence is chosen to provide a complementary analysis. It considers a detailed and fine level of granularity. The study of dependence clusters concerns dependence of all statements upon each other, while for predicate dependence, the results obtained refer to a semantically important aspect of the computation — the decision logic and consequent control flow of the program.

The primary contributions of the paper are as follows:

1. The paper introduces the concept of dependence anti pattern.

2. The paper shows how dependence analysis and visualisation can be used to identify dependence anti patterns.

3. The paper illustrates the the approach for six real world programs used by Daimler, showing how anti patterns help to identify possible problems for on–going evolution.

The remainder of this paper is organised as follows Section 2 presents background material on the dependence analysis and visualisation techniques used to make the paper self–contained. Section 3 introduces the concept of dependence anti pattern and gives several examples of dependence anti patterns. Section 4 briefly describes the programs studied. Sections 5 and 6 present the results of dependence analysis using dependence cluster visualisation and predicate dependence analysis. Section 7 presents related work, while Section 8 concludes.

2 Analysis Techniques Used

The dependence analysis an visualization techniques used in this paper are taken from previous work on predicate dependence [3] and dependence cluster analysis [2]. This section briefly reviews these techniques to make the paper self–contained. Both techniques are built on program slicing. A program slice extracts a semantically meaningful portion of a program, based upon a user–selected slicing criterion [23, 4].

2.1 Dependence Cluster Analysis

A dependence cluster is a set of program points (here taken to mean nodes of the Control Flow Graph (CFG)) that mutually depend upon one another. Any change to the computation represented at one point in a dependence cluster potentially affects the computations represented by all other points in the cluster.

It is possible to identify dependence clusters using slicing: those nodes having the same slice form a dependence cluster. However, an approximation is used for 'same slice' which is, not only more efficient to compute, but also leads to a useful visualisations for identifying clusters: the Monotone Slice-size Graph (MSG)[2]. Rather than testing if two SDG vertices have identical slices, the approach simply compares the slice size for the two vertices.

Monotone Slice-Size Graphs (MSG) have been developed to aid in the visual identification of dependence clusters [2]. Dependence clusters can be detected by seeking parts of the MSG where a large number of slices have the same size. This manifests itself as a 'plateau' in the graph as shown in Figure 1.

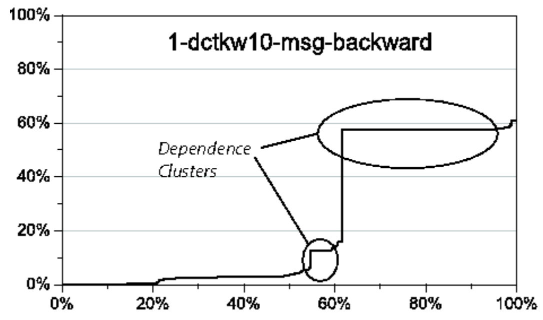

Figure 1. An Example MSG

2.2 Predicate Dependence Visualisation

Predicate dependence expresses the extent to which a particular predicate depends on variables in its scope. In order to visualize predicate dependence, three variables must be depicted:

1. Predicate Count: The number of predicates summarised by a data point.

2. Max-Parameters: The maximum number of parameters in scope (visible) at a predicate.

3. Parameters-Used: The number of parameters that affect a predicate according to the dependence analysis.

In these definitions "parameters" refers to formals and globals taken together. When only formals are being considered the terms Max-Parameters and Parameters-Used become Max-Formals and Formals-Used. Similar specialisation is applied when only globals are considered.

In a dependence bubble chart the horizontal axis denotes Max-Parameters, while the vertical axis denotes Parameters-Used. Thus, the line $y = x$ represents the worst case (a predicate can reference no more than the visible parameters). For reference, this line is drawn as the solid line in all dependence bubble charts. Also drawn is a dashed line that represents the linear trend (computed using a linear least squares fit). Finally, bubble size represents predicate-count.

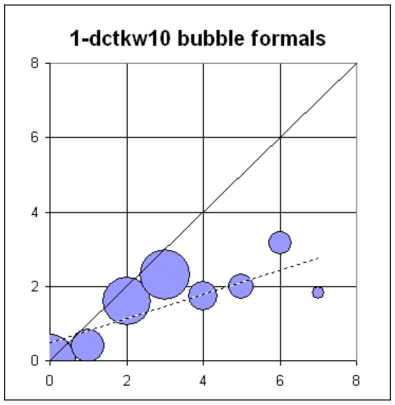

Figure 2. Dependence Bubble Chart Example.

For example, consider 1-dctkw10's dependence bubble chart[1] for formal parameters shown on Figure 2. The largest two bubbles summarize predicates in procedures with 2 and 3 formals respectively. The average number of formals used by all predicates from 1-dctkw10 with 2 formals in scope is around 1.5 (visually, these can be seen to depend upon an average of approximately 1.5 of the 2 available formal parameters).

3 Dependence Anti Patterns

A Dependence Anti Pattern is a dependence structure that can potentially cause problems. A Dependence Anti Pattern is defined by the name of the anti pattern, together with tell-tale dependence structure signs that signify the presence of the pattern (the signature of the anti pattern) and the reason why it may be problematic. This section lists 7 Dependence Anti Patterns. The list is by no means exhaustive and it is likely that other authors may be able to define other potential Dependence Anti Patterns. However, it is hoped that the list introduced here is sufficiently broad to give a flavour for the possibilities and to facilitate a set of realistic case studies in the following section.

[1] `dctkw10` is the name given to this program by Daimler.

Large Dependence Cluster (LDC)
Signature
A dependence cluster is a set of nodes all of which dependence upon one another [2]. All programs will contain some dependence clusters. Small clusters are not a problem. What constitutes 'large' will depend on the application.
Problems
The meaning of each node is dependent on the meaning of all, potentially increasing effort to comprehension and testing and making separation of concerns difficult.

Separable Formal Parameter (SFP)
Signature
A large number of formal parameters in a function, in which each predicate depends on few of these.
Problems
This is often an indicator of low cohesion. Especially when different (small) sets of formals are used with different functions.

Intense Formal Parameter (IFP)
Signature
A large number of formal parameters in a function, in which each predicate depends on many of these.
Problems
Given the limited human short=term memory, predicates that depend on more that a handful of formals have a negative impact on comprehension. This pattern also increases the complexity of testing (both by hand and using automated test-case generators).

Globals as Formal Surrogates (GFS)

Signature

Absence of formal parameter dependence combined with the presence of global variable dependence.

Problems

This pattern is often associated with poor function decomposition where is is difficult to identify the information flow in and out of functions where all function share common pools of globals.

Absent Predicate Dependence (APD)

Signature

A predicate with neither formal parameter nor global variable dependence.

Problems

Predicates that depend only on constants come in two forms. The first are non-problems. These occur in cases such as “for each letter in the alphabet” where the number of iterations is a known ‘universal’ constant. The second form occurs when one of more variables that should have been present in the predicate were forgotten. Perhaps left over from changes made in support of debugging or testing (*e.g.*, test stubs), such omissions can cause incorrect execution.

Separable Global Dependence (SGD)

Signature

A function with a large number of globals transitively defined or used, but containing some predicates that depend on relatively few.

Problems

Like SFP, given the limited human short-term memory, predicates that depend on more that a handful of globals have a negative impact on comprehension. Globals are less of a a problem, in terms of memory, because they are more likely to be tied to concepts stored in long-term memory. However the non-local nature of assignment to globals, make details understanding of predicates including a large number of globals a greater challenge. As with SFP, this pattern also increases the complexity of testing (both by hand and using automated test-case generators).

Intense Global Dependence (IGD)

Signature

A predicate with a relatively high dependence on global variables, compared to other predicates.

Problems

The comprehendability of such predicates is a challenge. Furthermore, depending on the condition, finding test data that exercises both branches can be difficult.

4 Analysis Subjects

The techniques described in the previous section were applied to the six programs supplied by Daimler for the purposes of this project. As shown in Table 1, the programs ranged from 3,298 to 9,165 lines as counted by the unix utility wc. In terms of actual non-comment non-blank lines of code the size ranged from 1,961 to 5,605 lines as seen in the columns headed ‘SLOC’ (source lines of code).

Two of the six programs (numbers 1 and 2) were obfuscated to protect against disclosure of the source code. However, the obfuscation is dependence–neutral and so the results presented are not affected by this obfuscation. Its only impact on the examples presented is the replacement of meaningful identifier names. Two of the programs were not obfuscated though they do come from Daimler development (these are program numbers 3 and 5). The remaining two programs (numbered 4 and 6) are open source program used by Daimler, which do not form a part of their production code portfolio.

5 Dependence Cluster Analysis

Figure 3 shows the MSGs of six programs studied. These programs all show plateaus in their MSGs although those in 3-netflow and 5-apkw2 are not as severe as the others. These plateaus provide the evidence that the programs contain dependence clusters.

Table 2 shows some general information regarding the dependence clusters of the six programs. This information includes the number of functions in each program, the number of functions in the largest dependence cluster, and the size of the largest dependence cluster, as the ratio of the number of program points in the cluster to the number of program points in entire program. Based on the 10% value as a threshold for a dependence cluster to be considered to be large [2], all programs except program 3-netflow contain at least one large dependence cluster.

Program `3-netflow` has a cluster that contains only 8% over the program. Program `5-apkw2` contains a dependence cluster cluster accounting for 13% of the program. Although the latter is slightly over the 10% level, these dependence clusters are considered to be relatively small compared to the others and were not studied further here in. This

Number	Subjects	Size(LOC)	SLOC	Description
1	1-dctkw10	9,165	7,537	obfuscated
2	2-srtrev1	5,087	4,046	obfuscated
3	3-netflow	4,391	2,672	Dynamic Data Functions
4	4-i2makw	6,899	5,605	Non Daimler code - Control Bank program
5	5-apkw2	3,298	1,961	Daimler ITT Experiments
6	6-ctbot	8,173	3,138	Non Daimler code - Robot simulator

Table 1. Experiment subjects.

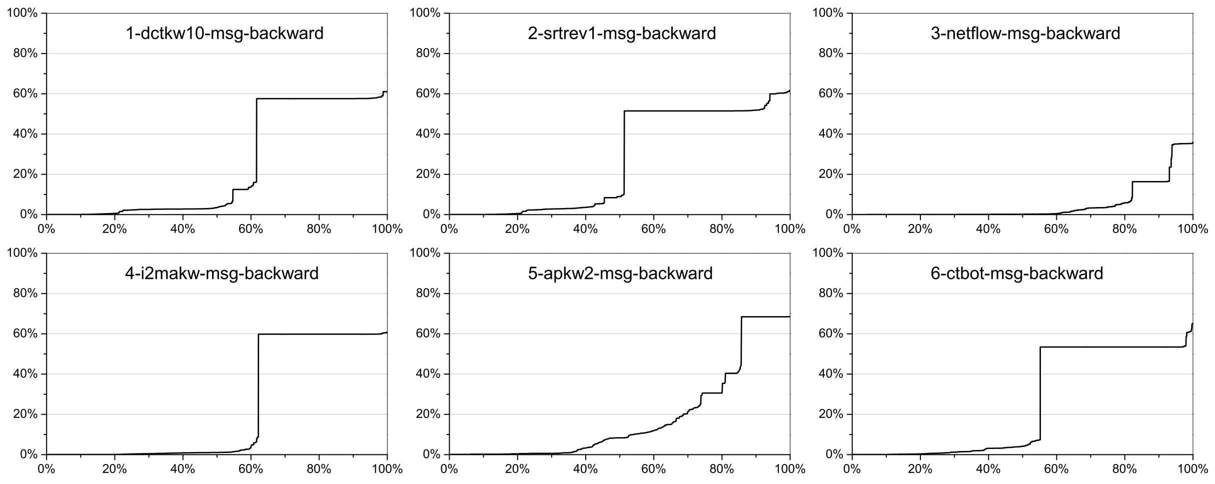

Figure 3. MSG-Backward

Subjects	Numbers of Functions	Numbers of Functions in DC	DC size
1-dctkw10	77	35	29%
2-srtrev1	66	27	32%
3-netflow	191	1	8%
4-i2makw	125	49	33%
5-apkw2	21	4	13%
6-ctbot	104	35	39%

Table 2. Dependence Clusters

leaves four programs with very large dependence clusters. The remainder of this subsection considers the dependence clusters for each of these four programs in more detail.

The MSG of `1-dctkw10` in Figure 3 and data in Table 2 indicate that the program `1-dctkw10` contains a large cluster, which is 29% of the whole program. There are total 77 user defined functions in `1-dctkw10`, of which 35 are involved in the large clusters. The largest cluster in the program `1-dctkw10` includes several sub-clusters. Once function (`i_d1533`) transitively calls many other functions. Inspection of this function's code reveals that its entire body is a for loop that calls the other functions. This causes much of each called functions to be part of the cluster.

The second program, `2-srtrev`, contains a large cluster, which is about 32% over the program. The program contains a function (`i_d106`) that transitively calls all other functions. Inspection of the function's code reveals that it contains four large for loops which capture all its called functions. In 15 of the functions (10 of which occur in the largest cluster), there is a prominent global variable that appears to be repeatedly defined and used. It might be supposed that this variable, being a prominent global variable, might play a large role in binding together dependence, thereby forming the large cluster.

In an attempt to verify this conjecture, certain global variables were replaced, with constants (when appearing as an r-value) and commented out (when appearing as an l-value), and then the MSG was recomputed. However, this somewhat aggressive replacement made no noticable difference to the MSG. We concluded that perhaps this is an example of a highly robust dependence cluster, suggesting that it occurs for deep seated and robust structural reasons.

The program `4-i2makw` contains a large cluster which contains 33% of the program. Most functions in this program are defined as `void functionname(void)`, *i.e.* most functions have no parameters and do not return a value, so the external effects of these functions are all communicated through global variables. Inspection of the code reveals that a total of 180 global variables are defined and used in most of the functions at control-points or in expressions. Source inspection finds that the program contains a `switch` statement the cases of which call the functions that participate in the dependence cluster.

Finally, the program `6-ctbot` contains a large cluster, which is about 39% over the whole program. Further investigation reveals that the function `bot_behave` has a pointer variable accessed by all 17 of the functions that call it and that these calls all use the pointer to update the value of a single global variable.

6 Predicate Dependence Analysis

Predicates are an important part of program functionality as they capture the logical flow of control of the program. For this reason, predicate dependence is worthy of special study in its own right. If a predicate depends on few formal parameters and few global variables, then it is likely to be easier to understand the role of the predicate in isolation. It is also easier to generate test data to cover the branches of the predicate, because there will be a smaller search space in which to locate suitable test data. Furthermore, changes to such a 'low dependence predicate' will have fewer potential influences, suggesting that such changes will be easier to perform. These three observations tend to suggest that high levels of predicate dependence are to be deprecated and motivates the study of predicate dependence for the suite programs under consideration in this study.

It must be stressed that the analysis shows what a predicate really depends upon in terms of the two influencing sources of dependence — formal parameters and global variables. However, in order for a programmer, maintainer, or tester to avail themselves of this information, it would be necessary for such a person to have access to the result of the dependence analysis. Such information can be produced by a tool for variable dependence analysis such as Vada [12].

In order to facilitate comparison between programs and between forms of predicate dependence analysis, the results are grouped together and presented in tabulated figures. The results presented in these figures can be compared to the results obtained from the suite of open source programs originally studies by Binkley and Harman [3].

6.1 Formal Parameter Dependence

The dependence bubble charts for formal parameter dependence of the six programs are depicted in Figure 4. A wide angle between the trend line (shown dotted in the figures) and the 45^o line (shown solid in the figures) indicates the potential for search space reduction for test data generation. It can be argued [3] that a large angle is a sign that the functions containing these predicates are less than fully cohesive. That is, such a wide angle means that a function has many predicates that fail to depend upon a large proportion of the formal parameters available to them. Such a function could be re-factored into smaller functions with fewer formal parameters and a greater degree of dependence within their logical structure. Such a re-factoring would have the goal of increasing the cohesion of the function so-created. It is also an indication of a lack of cohesion in the code prior to re-factoring.

It is immediately obvious that there are no programs in the Daimler set that exhibit such a large angle between formals used and formals available as those found in the first row of programs from the prior study (*e.g.*, that of copia and compress). This could be a sign that the functions from the Daimler suite are more cohesive. However, it offers less scope for reducing the number of formals that will need to be considered in order to generate test data from formal parameter values.

It is also striking that all of the programs in the Daimler suite have predicates that depend upon no formal parameters at all because they have none available to them. This is indicated by bubbles the centre of which lies on the origin. Such formal-free predicates are far less common and less pronounced in the prior study [3]. The phenomenon is particularly striking in the program 5-apkw2, where almost all predicates in the program depend on no parameters (and have none available).

This indicates that global variables are used to convey information to predicates in place of formal parameters. This has a saving on efficiency, but it may make the programs harder to understand and test. Understanding is impaired because the potential scope of a global is so much wider than that of a formal. The ability to generate test data is impaired because setting up values for globals is less convenient than simply calling a method or function with a chosen set of parameter values as an 'input vector'.

6.1.1 Observations and Insights gained

Bubbles	Predicate Count	Max Parameters	Average Parameters Used
A	6	7	1.83
B	110	7	5.61
C	279	0	0.00
D	136	0	0.00
E	4	5	1.00
F	207	1	0.09

Table 3. The parameters for Bubble A-F

Six bubbles are worthy of closer attention. These are marked as dark grey (rather than light grey) bubbles and labeled by the letter A-F in Figure 4. Table 3 shows Predicate Count, Max-Parameters and Average-Parameters-Used for these six bubbles. The rest of this

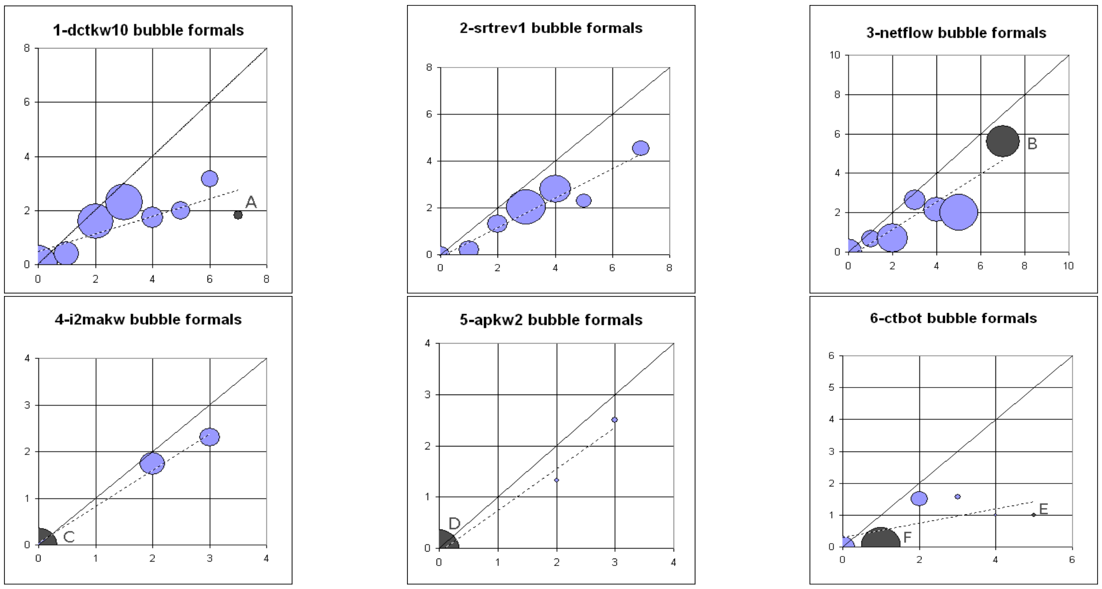

Figure 4. Formal Parameter Dependence bubble charts

subsection considers these six bubbles in more detail examining the properties of the code to which they correspond and the evidence they provide with regard to testing, comprehension and maintenance.

First, consider Bubble A from 1-dctkw10's dependence bubble chart for formal parameters shown on Figure 4. This bubble summarizes six predicates, all of which are located in procedures with 7 formals. These predicates are those that, for this program, have the the largest number of formals available to them. These six predicates all occur in a single function: function i_369. The maximum number of formals actually depended upon is 3, the minimum is 1, with an average of 1.83. This indicates that this function may contain logic that can be separated into different sub-functions, an example of Separable Formal Parameters (SFP).

Next, consider Bubble B from 3-netflow's dependence bubble chart for formal parameters shown on Figure 4. This bubble summarizes 110 predicates in procedures with 7 formals. Like the previous example, this is the largest number of formal parameters available to any predicate in the program. The average number of formals used by all predicates from 3-netflow with 7 formals in scope is 5.61. The 110 predicates occurred in 3 functions: 71 predicates in function NetFlow in NetFlow.c, 9 predicates in EvalSwitch in ProtocolFunctions.c and 30 predicates in function ReadTreeStructure in DynamicDataFunctions.c respectively.

Comparing the Bubble A in 1-dctkw10 and the Bubble B in 3-netflow, both have the same Max-parameters. However the Bubble B depends upon more parameters than Bubble A. This shows that analysis of dependence will be beneficial in reducing testing effort for the function i_369 in 1-dctkw10 but less so for NetFlow, EvalSwitch and ReadTreeStructure in 3-netflow, despite the fact that all four functions have the same number of formals available to them. Furthermore, despite the observation that NetFlow is likely to be the easiest program to understand and maintain, within it, the functions that are likely to be hardest to understand are NetFlow, EvalSwitch and ReadTreeStructure. This is an example of Intense Formal parameters (IFP).

Both the Bubble C in 4-i2makw and the Bubble D in 5-apkw2 include a large number predicates (279 and 136 respectively). These predicates do not depend upon any formal parameters. A further analysis of global variable dependence for these predicates reveals that there are global variables available to all these predicates, with most depending on some of these global variables: Of the 279 predicates summarised by Bubble C, only 19 depend upon no global variables (as well as depending upon no formal parameters). This suggests possible instances of the Globals as Formal Surrogates (GFD) anti pattern. Of these 19, 16 predicates are for loop conditions, the other 3 predicates are if conditions.

The source code for one (typical) for loop is

```
for( LoopCounter = 0;
     LoopCounter < I2MA_CYL_ARRAY_SIZE;
     LoopCounter++).
```

Here, I2MA_CYL_ARRAY_SIZE is a constant of 5. The condition neither depends upon formal parameters nor global variables. However, this is typical for a for loop;

semantically, it has lower and upper bounds that are determined by compile-time constants. As such, it is unlikely for a for loop to depend upon anything. Indeed, where a for loop does depend upon either formals or globals, it is extremely likely that such a loop is really a while loop, masquerading as a for loop.

The other 3 predicates of the 19 that have no dependence are if conditions. This is an instance of the Absent Predicate Dependence (APD) anti pattern. However, upon further inspection, it becomes clear that these predicates depend upon the return value of the function LimpinIsSetFor. The only statement in the function LimpinIsSetFor is "return 0". Further inspection uncovered the comment:

"This is a stub function - in the original system this is a signal to a parallel subsystem"

In this way the anomalous predicate dependence for these three predicates has shown up the presence of uninstantiated stubs in the code.

The Bubble F in 6-ctbot provides another example of a set of predicates that have little dependence on formals. However, in this case, there are very few (1) formals for these predicates to depend upon. In total, there are 207 predicates and these 207 only depend upon an average of 0.09 of the single formal parameter available. In fact, a closer analysis reveals that only 19 of 207 predicates depend upon 1 formal parameter, while the others depend upon none. A similar global variable dependence analysis indicates that all predicates except 2 depend on global parameters. This provides evidence for both the Absent Predicate Dependence (APD) and the Globals and Formal Surrogates (GFS) anti patterns.

The Bubble E in 6-ctbot is another example with large Max-parameters but a few used. This bubble includes 4 predicates in procedures that depend upon only 1 of 5 formal parameters. All 4 predicates occur in two functions with 5 formal parameters, command_write and command_write_data.

6.2 Global Variable Dependence

The dependence bubble charts for global dependence of the six programs are depicted in Figure 5. Eight bubbles are worthy of closer inspection. These are marked as black bubbles (rather than light grey) and labeled by 1-8 in Figure 5. Table 4 shows Predicate Count, Max-Parameters and Average-Parameters-Used for the these eight bubbles.

First, consider Bubble 1 from 1-dctkw10 dependence bubble chart for global variables. This bubble summarizes 9 predicates in procedures with 157 globals available to them. The number of globals used for each of these predicates from 1-dctkw10 are 1, 1, 98, 98, 98, 1, 1, 81, 0 respectively, and the average number of globals used by all predicates is 42.11.

Bubbles	Predicate Count	Max Parameters	Average Parameters Used
1	9	157	42.1
2	5	224	4.0
3	5	191	6.0
4	30	25	12.5
5	7	187	2.0
6	1	82	0.0
7	7	91	76.0
8	8	139	2.0

Table 4. The parameters for Bubbles 1-8

Clearly, it can be seen that four predicates depend upon a large number of global variables. Inspection of the code reveals that all nine predicates occur in function i_1732, which has no formal parameters. Clearly in this function, globals are being used to communicate values to and from the function, an example of the Separable Global Dependence (SGD) pattern. Those predicates that depend upon only one such global would be better written as calls to simple functions that return the value true/false and take the single global as a parameter (perhaps modifying it if necessary). This would make these predicates far easier to understand, since they would clearly depend upon a single value rather than 157!

Recall that in Section 2.1, the dependence cluster analysis indicated that the program `1-dctkw10` contains a large cluster; one which consumed 29.10% of the whole program. A further inspection of the source of this large dependence cluster reveals that the four predicates that depend on a large number of globals are all in the large cluster, while the other five predicates with few globals depended are not. This provides strong evidence that the use of large number of global variables in source could result in a large cluster. It also suggests that the four predicates that depend on large numbers of globals should be factored out and carefully considered in order to determine why they have such a high dependence.

Bubble 4 in 3-netflow's dependence bubble chart for global variables is shown on Figure 5. This bubble shows 30 predicates in 3-netflow depending upon an average of 12.47 of the 25 global variables. Inspection of the code reveals that all 30 predicates occurred in function ReadTreeStructure. This is an example of the Intense Global Dependence (IGD) pattern. The high level of global dependence and the name of the function suggest that the tree structure in question may be a global data structure. If it is possible to decompose this function, then it will be easier to test (if the function is decomposed into smaller functions, each of which only touches those parts of the global structure that are needed).

Bubble 7 summarizes seven predicates in 6-ctbot with 91 global variables available. Each predicate in this bubble

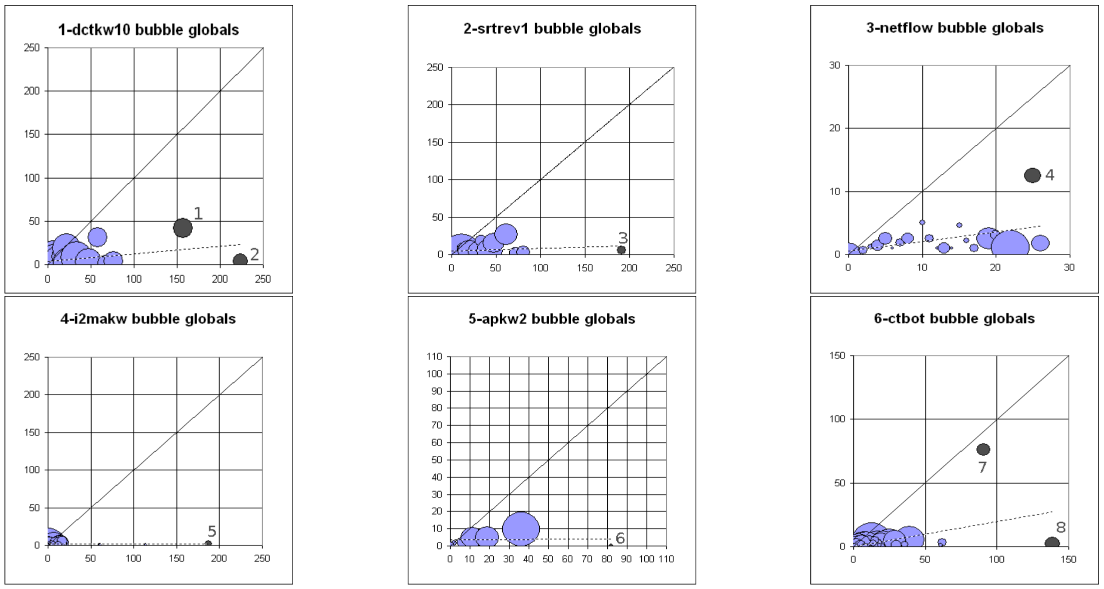

Figure 5. Global Variable Dependence bubble charts

depends upon 76 of 91 global variables. These seven predicates all occur in function bot_behave. The fact that these predicates depend (really depend that is, not merely appear to depend) on such a large number of global variables indicates that this function's logical structure will be very hard to understand and test. This provides another example of Intense Global Dependence (IGD).

The reader may wonder about bubbles 2, 3, 5, 6 and 8. These are examples of predicates that only depend upon a few global variables, but which have large number of globals available to them. The predicates in these five bubbles all occur in function main. This function typically pulls together the strands of the program, so it will have a large number of globals available to it. It would not be abnormal for main to contain predicates that depend on very few of the total globals, which explains most of these bubbles.

However, all such bubbles cannot be completely dismissed. Bubble 6 includes one predicate that depends upon *no* globals and no formals; an example of Absent Predicate Dependence (APD). The corresponding source code is "while(1)"; a never ending loop. Similarly, one of predicates in the Bubble 8 is the same, except that the corresponding source is "for(; ;)".

7 Related Work

Harman and Binkley introduced the idea of dependence clusters, showing that they occur in real world programs, though they did not explicitly identify them as Dependence Anti Patterns [2]. This paper provides further evidence that dependence clusters are highly prevalent in real production code. The six Dependence Anti Patterns considered in the present paper and the concept of Dependence Anti Patterns have not been previously considered.

However, work on the general concept of patterns is very well known from the work of the gang of four [10] and has previously been suggested as a technique for enhancing software evolution [1]. Our work differs, because it is based on dependence structure patterns, rather than design patterns, but our approach is inspired by previous work on design patterns.

Design patterns capture the templates of design ideas that have been considered to work well in certain engineering scenarios by a large body of experienced engineers. This seminal work on design patterns led to the consideration of anti patterns; templates of design structures that are believe not to work well and which should be avoided. However, the present paper is the first to consider dependence structures as potential patterns. While it is hard to define what a 'good' dependence structure should look like, it is comparatively easy to identify dependence structures that denote potential problems; these are the dependence anti patterns considered in the present paper.

In software evolution, the idea of templates of design and patterns that can be identified, isolated, and studied has received a lot of attention. Often the artifacts studied go under different names such as work on identification of program plans [20, 25, 21] and program clichés [22, 9]. However this previous work in identification of programming style templates such as plans and clichés has not considered dependence structures of the form considered in the present paper.

Perhaps the closest work to that presented here considers work on identification if induction variables [24], and automated parallelization [13, 16], since these are a kind of program construction that is identified, in part, by a dependence signature. However, induction variables are individual variables with the goal of informing loop analysis, while work on automated parallelization is concerned with teasing apart dependence structures to facilitate increase opportunities for parallelism. By contrast, the dependence structures considered in the present paper are larger–grained, potentially involving many variables and program points and their interactions, while the goal is to identify potential problematic dependence structures.

8 Conclusion and Future Work

This paper has introduced the concept of dependence anti patterns, showing how these anti patterns can be defined and located using dependence analysis and visualization. The paper argues that the study of dependence anti patterns will provide useful insights into possible problems and issues for on–going maintenance. Evidence for this claim was provides by the presentation of six case studies, which show the presence of these anti patterns in real production code.

References

[1] F. Arcelli and L. Cristina. Enhancing software evolution through design pattern detection. In *Software Evolvability (Evol '07)*, pages 7–14, Paris, 2007. IEEE Computer Society Press.

[2] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In 21^{st} *IEEE International Conference on Software Maintenance*, pages 177–186, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[3] D. W. Binkley and M. Harman. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Transactions on Software Engineering*, 30(11):715–735, 2004.

[4] D. W. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[6] N. I. Churcher and M. J. Shepperd. Comments on 'A metrics suite for object oriented design'. *IEEE Transactions on Software Engineering*, 21(3):263–265, Mar. 1995.

[7] N. E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman and Hall, 1990.

[8] N. E. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, 1994.

[9] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. A cliche-based environment to support architectural reverse engineering. In *International Conference on Software Maintenance (ICSM'96)*. IEEE Computer Society Press, 1996.

[10] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[11] M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[12] M. Harman, C. Fox, R. M. Hierons, L. Hu, S. Danicic, and J. Wegener. Vada: A transformation-based system for variable dependence analysis. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 55–64, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[13] Y. Lee and B. G. Ryder. Effectively exploiting parallelism in data flow analysis. *The Journal of Supercomputing*, 8(3):233–262, Nov. 1994.

[14] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[15] M. C. Paulk, B. Curtis, E. Averill, J. Bamberger, T. Kasse, M. Konrad, J. Perdue, C. Weber, and J. Withey. Capability maturity model for software. Technical Report CMU/SEI-91-TR-24 ADA240603, Software Engineering Institute (Carnegie Mellon University), 1991.

[16] C. Ryan. *Automatic re-engineering of software using genetic programming*. Kluwer Academic Publishers, 2000.

[17] M. J. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):177–188, 1988.

[18] M. J. Shepperd. *Foundations of software measurement*. Prentice Hall, 1995.

[19] M. J. Shepperd and D. C. Ince. A critique of three metrics. *Journal of Systems and Software*, 26:197–210, 1994.

[20] E.-S. Tan and H. G. Dietz. Abstracting plan-like program information: A demonstration. In *Proceedings of the International Conference on Software Maintenance (ICSM 1994)*, pages 262–271. IEEE Computer Society Press, Sept. 1994.

[21] A. van Deursen, A. Quilici, and S. Woods. Program plan recognition for year 2000 tools. *Science of Computer Programming*, 36(2-3):303–324, 2000.

[22] R. C. Waters. Cliche-based program editors. *ACM Transactions on Programming Languages and Systems*, 16(1):102–150, Jan. 1994.

[23] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[24] M. Wolfe. Beyond induction variables. *ACM SIGPLAN Notices*, 27(7):162–174, July 1992.

[25] S. Woods and A. Quilici. Some experiments toward understanding how program plan recognition algorithms scale. In *IEEE International Working Conference on Reverse Engineering (WCRE'96)*, pages 21–30. IEEE Computer Society Press, 1996.