Enhanced Dependency Structure Matrix for Moose

Alexandre Bergel, Stéphane Ducasse, Jannik Laval, Romain Peirs
RMoD Team, INRIA, Lille, France
firstname.lastname@inria.fr

Abstract

Dependency Structure Matrix (DSM), an approach developed in the context of process optimization, has been successfully applied to identify software dependencies among packages and subsystems. It exists a couple of algorithms to help organizing the matrix in a form that reflects the architecture and highlights patterns and problematic dependencies between subsystems. However, the existing DSM implementations often miss some important information in their visualization to fully support a reengineering effort. In this paper we enhanced DSM with enriched cell contextual information by showing information (i) about the kinds of references made (inheritance, class accesses..), (ii) the proportion of entities (classes/methods) doing references, (iii) the proportion of entities been the target of the references. We distinguish independent cycles and stress the cycles using coloring information. This work has been implemented on top of the Moose open-source reengineering environment and Mondrian. It has been applied to non-trivial case studies such as the Morphic UI frameworks available in Squeak an open-source Smalltalk.

1 Introduction

Understanding the structure of large applications is a challenging but important task. Several approaches provide information on packages and their relationships, by visualizing software artefacts, metrics, their structure and their evolution. Software metrics can be somehow difficult to understand since they are dependent on projects. Distribution Map [3] alleviates this problem by showing how properties are spread over an application. Lanza et al. [4] propose to recover high level views by visualizing relationships. Package Surface Blueprint reveals the package internal structure and relationships among other packages. A package blueprint is structured around the concept of surface, which represents the relationships between the analyzed package and its provider packages [5].

Dependency Structure Matrix (DSM) is an approach originally developed for process optimization. A variety of algorithms exist to organize the matrix in a form that reflects the architecture. These algorithms highlight patterns and problematic dependencies among tasks. It has been successfully applied to identify software dependencies [7, 10, 11]. MacCormack *et al* [8] have applied the DSM to analyze the value of modularity in the architectures of Mozilla and Linux.

While DSMs are a proven solution to reveal software structure, DSMs have weaknesses too. They lack providing certain information when computed and rendered. DSM current implementations lack a fine grained information without losing their overview ability. Certain algorithms produce blurry cycles with the power of adjacency matrix method in which independent cycles are merged. Moreover detected cycles are not focused on an entity and lack of support for class extension.

Our contribution addresses such weaknesses. We enriched cell contextual information by showing information (i) about the kinds of references made (inheritance, class accesses..), (ii) the proportion of entities (classes/methods) doing references, (iii) the proportion of entities been the target of references. We provide hints at packages introducing wrong references. We distinguished independent cycles and stress the cycles using coloring information. In addition, we distinguish true cycles from independent cycles within the same level. Finally we offer an entity-focused oriented view of level cycle.

2 DSM Limitations

We identified a number of limitation for DSMs: merging independent cycles (Section 2.1), lack of fine grained-overview (Section 2.2), cycles not focused on an entity (Section 2.3) and lack of support for class extension (Section 2.4).

2.1 Blurry Cycles with the Power of Adjacency Matrix Method

A way to identify cycle in DSM is to use the technique based on powering the adjacency matrix. The principle of this approach is to raise a binary DSM to its n-th power to indicate which elements can be traced back to themselves in n steps; thus constituting a cycle [12]. However, the indicated elements do not automatically belong to the same cycle. Indeed, it can exist several cycles with the same number of steps and the power of the adjacency matrix method cannot differentiate these different cycles, so we have blurry cycles.

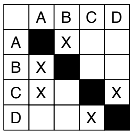


Figure 1. A DSM

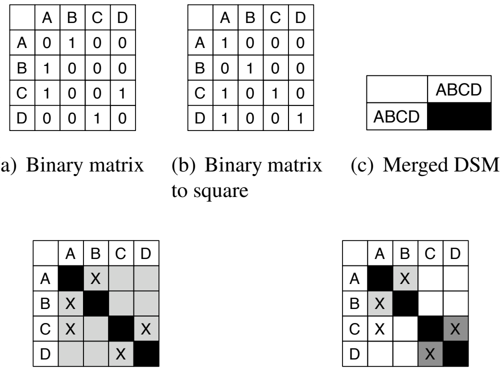
On Figure 1, we see that the elements A and B constitute a direct cycle and the elements C and D constitute another one. But if we raise the binary DSM (Figure 2(a)) to the square by applying the adjacency matrix technique, a non-zero value appears in the diagonal for every element (Figure 2(b)). These non-zero values mean that any of the elements A, B, C and D are involved in at least one direct cycle but these non-zero values do not show what these direct cycles are made of. Moreover, with the partitioning algorithm based on powering the adjacency matrix, we merge these 4 elements together (Figure 2(c)) which means that in the partitioned matrix these elements will appear as one cycle (Figure 2(d)) - note that the grey zone represent the cycles. So, the partitioned matrix provides a wrong information by indicating a unique cycle (the grey area in Figure 2(d)) whereas the matrix should show two direct cycles as shown in Figure 2(e).

So, the power of adjacency matrix method does not allow us to determine precisely the different cycles. However, combining this algorithm with a path searching method identifies correctly all the different cycles.

Notice that the DSM software Lattix does not use the power adjacency partitioning algorithm but is using reachability matrix method.

2.2 Lack of Fine-grained Information

A traditional DSM offers a general overview but lacks from providing precise information about the situation it describes. We identify two kinds of weaknesses: lack of dependency causes and lack of dependency impacts.

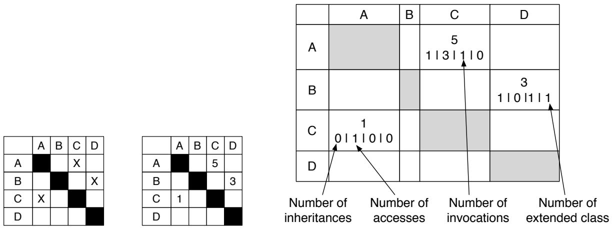


(a) Binary matrix (b) Binary matrix to square (c) Merged DSM

(d) Partitioned DSM with adjacency matrix method (e) Ideal partitioned DSM with two distinct cycles

Figure 2. Limitation of the power of adjacency matrix method

Lack of Causes. Dependencies can be due to several sources: class direct accesses, class extensions (see Section 2.3), inheritance relationships and method invocation. Fixing cycles may be different based on the source of linking *e.g.,* changing a direct reference to a class is often simpler than changing an inheritance relationship. That is why, it is not enough to just indicate the dependencies in a DSM with a *X* (Figure 3(a)) or even with a number representing the number of dependencies that exist (Figure 3(b)). Indeed, indicating at least a number for each kind of dependency (Figure 3(c)) gives a more fine-grained information and so supports a better understanding of the situation.



(a) DSM with marks (b) DSM with numbers (c) DSM with dependency kinds

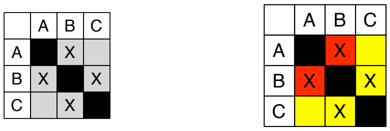
Figure 3. Examples of references in a DSM

Lack of Impacts. In addition, the information provided often lacks important information and does not give an idea of potential problems. For example, knowing that a package has 99 references to another one is a valuable information. Such references could be done by 20 or 3 classes and these 99 references could refer to a small subset or a large number of classes. The same remark can be done for methods instead of classes. But knowing that these references originate only from 3 classes and 15 methods and that only 4 classes and 6 methods are referenced by these 99 dependencies makes a large difference. In addition, having access to this information without having to look at every class in the concerned packages clearly supports the software comprehension.

2.3 Cycles Not Focused on an Entity

Cycles are shown in the context of complete system and the ordering within a level is the result of the partitioning algorithm. It makes it hard to understand the cycle (and not the level) in which a given package is involved. In particular when cycles inside the same level are merged, we do not obtained an accurate information.

As an illustration, consider a package A involved in a direct cycle with the package B and this package B is involved in a direct cycle with the package C (Figure 4(a)). We therefore have A in a cycle that also includes C but the length of this cycle is not the same that the length of the cycle between package A and package B. This is why it is a valuable information to see the difference between the lengths of the cycles (Figure 4(b)) since the used methods to break them will not be the same. In Figure 4(b) direct cycles started from the entity A are displayed: yellow shows the direct cycles, then red the second level.



(a) Cycles not focused on an entity

(b) Cycles focused on an entity

Figure 4. The importance of cycles focused on an entity

2.4 No Support for Class Extensions

A class extension is a method that is defined in a package, but whose class is defined in another package [1, 2]. Class extensions exist in Smalltalk, CLOS, Objective-C and now partly in C#. They offer a convenient way to incrementally modify existing classes when subclassing is inappropriate. Indeed, extending by subclassing may be inappropriate because that specific class is referred to, but, one cannot modify the source code of the class in question. A class extension can then be done to that specific class.

3 Enhanced DSM

Our apparoch takes into account the previously described limitations and adds some functionalities that does not seem to be offered by DSM software like Lattix [9]: (i) enriched contextual cell information (Section 3.1), (ii) isolating independent cycles and coloring information (Section 3.2), and (iii) entity focused oriented view and level cycle coloring (Section 3.3).

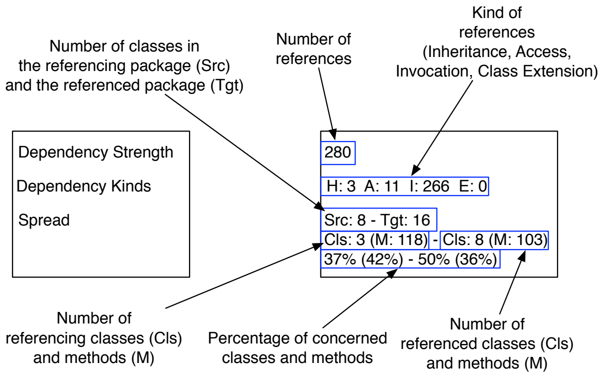


Figure 5. DSM cell with enriched contextual information

3.1 Enriched Contextual Cell Information

As described in the previous section, current DSM software do not provide a fine-grained information within the overview offered by the matrix. We address this issue in our enhanced DSM by enriching the cell contents (Figure 5). The enriched contextual cell information shows the following information:

1. *Strength of the dependency.* On the first line, we show the number of references from a package to another one. This number gives us the strength of the link which exists between these packages but give no more information.

2. *Dependency Kinds.* The second line shows the kinds of reference (Inheritance (H), Access (A), Invocation (I) and class Extension (E)) and how many references there are for each. These numbers give us a coarse grained information about the relationships between packages.

3. *Source and Spread of the Dependency.* Finally, we show how the references are spread in and to the packages. We give the numbers of classes in both packages

on the third line. This information gives us an idea of the size of the packages involved in the dependency. On the fourth line, we indicate for the package doing the reference and the package been referenced the number of classes and methods that are actually doing the references or been referenced. The last line express the same idea but with percentages. This information is important because it shows us two things: the importance of the classes doing the references and the spread of such references on the target package. We can get an idea of the spread and flow of the references. For example, whether it is a small part of a package or a big one which is source of the dependencies and approximates the necessary engineering effort to remove these dependencies.

This enriched contextual view supports the identification of situation by just glancing over the matrix.

3.2 Enhanced Cycle Detection

Identifying cycles is important in large entangled software. In the context of DSM, cycle detection is often realized using a path searching algorithm [6]. This algorithm allows one to detect separately every independent cycle. Indeed, as explained in Section 2.1, two packages, A and B, may be in a direct cycle and packages C and D in another direct cycle. In this case, these two cycles are identified with path searching algorithm, which cannot be achieved with an algorithm based on the powers of adjacency matrix. Our approach enhances the traditional matrix by proposing: cycle distinctions (Section 3.2), cycle nesting identification (Section 3.2), and hints for cycling fixing (Section 3.2)).

Cycle Distinctions. Our approach distinguishes independent cycles. It is based on the path searching algorithm [6]. With the path searching method, 2 independent cycles are separately detected and can so be isolated from each other in the DSM (Figure 6).

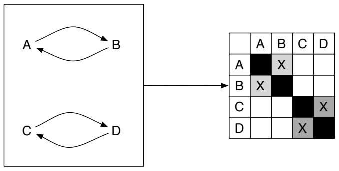
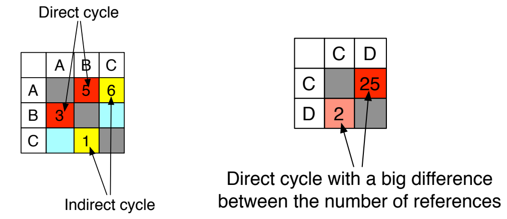


Figure 6. Independent cycles distinction

Cycle Nesting. We added color information in DSM cells to give information about cycles. Indeed, as shown on Figure 7(a), after partitioning, DSM cells involved in a cycle have a yellow or red color. The yellow color means that the 2 concerned packages (a DSM cell is the intersection of 2 packages) are involved in an indirect cycle (a cycle with more than 2 elements) while the red color means they are involved in a direct cycle. The pale blue square frames the whole of packages which are in cycle (it is visible in Figure 9), contrary to the white lines and colomns which represent packages with no cycle.

Cycle Fixing Hints. In addition, in the case of a direct cycle, if there is a large difference between the number of references between the 2 packages (by default, the ratio is 3), the cell that contains the least references has a light red color (Figure 7(b)). This information allows the user to focus his attention on the references which should be solved a priori in a first time. As shown in Figure 9, such hint is really useful to spot packages potentially introducing cycles.


Direct cycle
A B C
A
B
C
5
6
3
1
Indirect cycle
C D
C
D
25
2
Direct cycle with a big difference
between the number of references

(a) Cycles representation

(b) Cycle fixing hints

Figure 7. Direct and indirect cycles coloring

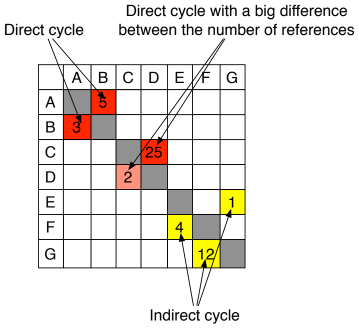

Direct cycle
Direct cycle with a big difference
between the number of references
A B C D E F G
A
B
C
D
E
F
G
5
3
25
2
1
4
12
Indirect cycle

Figure 8. Cell color definition

Figure 8 illustrates how to interpret the colored boxes. It shows that there are two direct cycles: between A and B and between C and D since they are red. The cycles are distinct since there is no red box on the A B lines coming from the C D columns. We see that D may be the source of problems for the cycle with C. Then there is an indirect cycle between E F and G since there is no red between them.

A Concrete Example. We applied our approach to the Morphic framework of Squeak. Note that it was never packaged in a modular way, hence showing a lot of cyclic dependencies. You can observe in Figure 9 a large cycle represented in pale blue. Without any color, this would be difficult to quickly distinguish direct and indirect cycles. Indeed, direct and indirect cycles would not clearly appear, so the observer could not easily separate them. But with colored cells, at first glance, identifying direct and indirect cycles is easy. In addition, you can focus your attention on the light red cells because they indicate what are the cells which should probably disappear to eliminate the direct cycles.

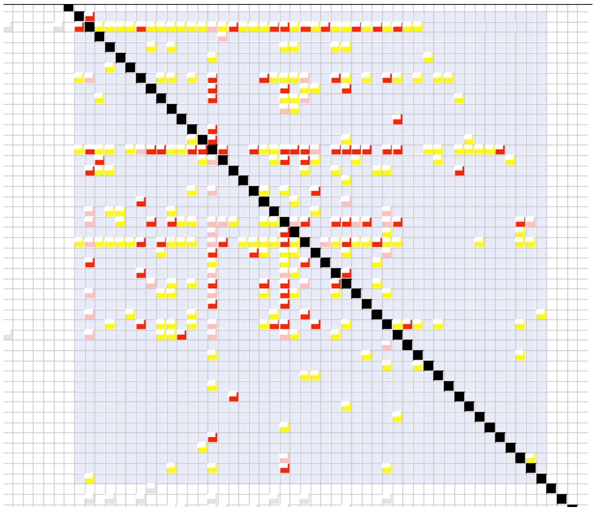
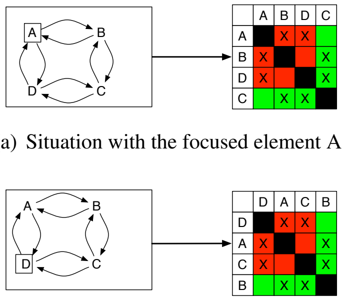


Figure 9. Example of cycle detection

3.3 Entity-focused Cycle Centric View

We have seen in the previous section that our approach isolates independent cycles and add information to them by coloring cells to distinguish direct and indirect cycles. However, if it is easy to see the different cycles which exist into a given layer when there are a few number of packages, it is much more difficult with a large number of packages. Indeed, you do not easily know if the length of a cycle between 2 packages is short or long. This is particularly complicated since an element can belong to several cycles which do not have the same length. It is difficult to show the length of cycles to which belongs an element.

To solve this problem, we added the notion of focused element. Indeed, the length of a cycle between an element and another one is relative to the first one. So, we represent the length of the cycles relative to a chosen element that we call the focused element. For example, in Figure 10, you can see that relative to the element A, the elements B and D are involved in a direct cycle with A; and the element C is in an indirect cycle with A (Figure 10(a)). But if you focus your attention on the element D, that are the elements A and C that are involved in a direct cycle with D and the element B which is in an indirect cycle with D (Figure 10(b)).



	A	B	D	C
A	■	X	X	🟩
B	X	■	🟥	X
D	X	🟥	■	X
C	🟩	X	X	■

(a) Situation with the focused element A

	D	A	C	B
D	■	X	X	🟩
A	X	■	🟥	X
C	X	🟥	■	X
B	🟩	X	X	■

(b) Situation with the focused element D

Figure 10. Visualization of the different cycles relative to the focused element

3.4 Cycle Level

So, to help the visualization of the different cycles in which your focused element is involved, we introduce a color information for every cycle level. Indeed, as shown in Figure 11, the elements involved in the first cycle level with the focused element (*i.e.,* the elements which are in cycle with the focused element and which the length of the cycle is the shortest) are in red color. This process is repeated for every element which is in cycle with the focused element: the elements which belong to a same cycle level are in a same color. Thanks to that color information, first, you can easily see whether 2 elements are close in a cycle or not and second, you can count the number of elements in every cycle level.

4 Conclusion

This paper enhances Dependency Structure Matrix (DSM), an approach which identifies dependencies between software packages. A DSM renders references between packages in a matrix-like table. In addition, several algorithms already exist to reorder the matrix and so reveal the software architecture.

However, a traditional DSM does not provide enough information about the references between the packages and about the cycles which can exist between them. This paper addresses such limitations by improving the information contained in a DSM and enhancing the way a DSM is

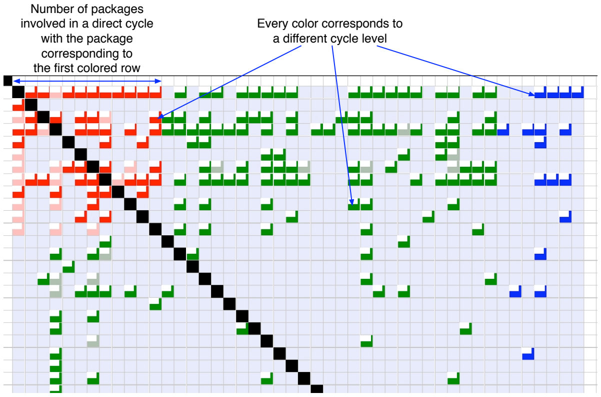


Figure 11. Visualization of different cycle levels

rendered. First, the cell contents is enriched with the kinds and spread of the references as well as with the number of classes involved. Second, several colors are used to distinguish direct and indirect cycles.

Thanks to these improvements, understanding and analysis of software applications are easier and thus faster. However, in future work, it would be very helpful to be able to see directly in the matrix the consequences of the changes brought to the application without having to change the code and may be from that a code generator could make itself the changes into the code from the changes realized into the matrix.

References

[1] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[2] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. *Journal of Computer Languages, Systems and Structures*, 31(3-4):107–126, Dec. 2005.

[3] S. Ducasse, T. Gîrba, and A. Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[4] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.

[5] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, pages 94–103, 2007.

[6] D. Gebala, S. Eppinger, and M. Cambridge. Methods for analyzing design procedures. *Design Theory and Methodology*, 1991.

[7] A. Lopes and J. L. Fiadeiro. Context-awareness in software architectures. In *Proceeding of the 2nd European Workshop on Software Architecture (EWSA)*, volume 3527 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2005.

[8] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.

[9] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[10] D. Steward. The design structure matrix: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, 28(3):71–74, 1981.

[11] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE 2001*, 2001.

[12] A. Yassine, D. Falkenburg, and K. Chelst. Engineering design management: an information structure approach. *International Journal of Production Research*, 37(13):2957–2975, 1999.