Cycles Assessment with CycleTable

Jannik Laval, Simon Denier, Stéphane Ducasse

Cycles Assessment with CycleTable

Jannik Laval, Simon Denier, Stéphane Ducasse

RMoD Team
INRIA - Lille Nord Europe - USTL - CNRS UMR 8022
Lille, France
{jannik.laval, simon.denier, stephane.ducasse}@inria.fr

***Abstract*—Understanding the package organization of a large application is a challenging and critical task since it allows developers to better maintain the application. Several approaches show in different ways software structure. Fewer show modularity issues at the package level. We focus on modularity issues due to cyclic dependencies between packages. Most approaches detect Strongly Connected Components (SCC) in a graph of dependencies. However, SCC detection does not allow one to easily understand and remove cyclic dependencies in legacy software displaying dozens of packages all dependent on each other.**

This paper presents i) a heuristic to focus on *shared* dependencies between cycles in SCC and ii) CycleTable, a visualization showing interesting dependencies to efficiently remove cycles in the system. This visualization is completed with enriched cells, small views displaying the internals of a dependency [LDDB09]. We performed i) a case study which shows that the shared dependency heuristic highlights dependencies to be removed, and ii) a comparative study which shows that CycleTable is useful for the task of breaking cycles in a SCC compared to a normal node-link representation.

***Keywords*-software visualization; reengineering; cycle; package; dependency**

Note for the reader: this paper makes heavy use of colors in the figures. Please obtain and read a colored version of this paper to better understand the ideas presented in this paper.

I. Introduction

It is frequent to have large applications structured over many packages. Packages are units of reuse and deployment: a package is built, tested, and released as a whole as soon as one of its classes is changed, or used elsewhere [Mar00]. Then modularity is as important at package level as at class level (if not more). Modularity implies that releasing a new package will only impact the dependent packages in the building chain. Cycles between packages have a high impact on the modularity of the application. Indeed, a cycle in the package dependency graph implies all packages in the cycle to be built, tested, and released together as they depend on each other. Martin [Mar00] proposes the Acyclic Dependencies Principle (ADP), which states that there should not be any cyclic dependency between packages.

While it is easy to detect and correct cyclic dependency as soon as they arrive, the problem is more difficult in legacy software where no cycle assessment has been performed on the go. Then dependencies often form one unique Strongly Connected Component (SCC) where all packages depend on each other. It is cumbersome to understand the interweaving of dependencies and difficult to devise an efficient plan for breaking cycles.

In a previous work [LDDB09], we propose eDSM which enhances Dependency Structural Matrix (DSM) for a better understanding of cycles at the package level. eDSM shows Strongly Connected Components and highlights cycles between two packages in the SCC. However, eDSM is not adapted when one cycle involves more than two packages.

We devise a new approach based on the decomposition of one SCC into a set of "minimal cycles". "Minimal cycles" are simple cycles containing a few nodes and are thus easy to understand and fix. Together the set of minimal cycles cover all dependencies in the SCC and allows the engineer to assess a minimal plan to remove all cycles in the system.

In this paper, we present a new heuristic to break cycles named "shared dependencies" and a new visualization, called CycleTable, entirely dedicated to cyclic dependency assessment. The heuristic of "shared dependencies" comes from the decomposition of the SCC into minimal cycles. Often minimal cycles are intertwined together so that one dependency is shared by the cycles. Removing such a shared dependency breaks involved cycles. CycleTable layout displays all cycles at once and shows how they are intertwined through one or more shared dependencies. CycleTable combines this layout with the enriched cells (eCell) of eDSM [LDDB09] to present details of dependencies at class level, which allows the engineer to assess the weight of the dependency. This work is implemented on top of the Moose software analysis platform.

Section II introduces the background and the challenges of cycle analysis with the traditional node-link representation of graphs and with DSM. Section III and Section IV explain CycleTable layout and usage. Section V presents enriched cells in CycleTable and discusses on a sample case the criteria to break cycles as highlighted by the visualization. Section VI presents some validations based on a case study and a comparative study. Section VII lists related work and Section VIII concludes the paper.

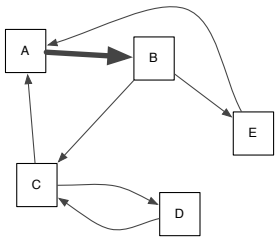

Figure 1. Sample SCC.

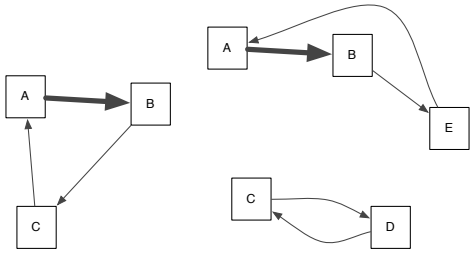

Figure 2. Sample SCC (Figure 1) decomposed into three minimal cycles. The thick arrow represents a *shared dependency*

II. Cycle understanding problems

In this section, we present important points related to cycle understanding and which methods exist to fix them.

A. Definitions

Definition 1: A Strongly Connected Component (SCC) is the maximal set of nodes (here, packages) that depend on each others. In Figure 1, all nodes are in a single SCC.

Definition 2: A cycle is a circular dependency between two or more packages. We distinguish two types of cycles:

- *direct cycle*. It represents a cycle between two packages. In Figure 1, C and D are in a direct cycle because there is one dependency from C to D, and one dependency from D to C.
- *indirect cycle*. It represents a cycle between more than two packages. In Figure 1, A, B and C are in indirect cycle. A, B and E are also in indirect cycle.

Definition 3: A *minimal* cycle is a cycle with no node (here no package) appearing more than once when listing the sequence of nodes in the cycle. In Figure 2, A-B-E and A-B-C are two different minimal cycles, but A-B-C-D-C is not because C is present twice. A-B-C-D-C can be reconstructed with the two minimal cycle A-B-C and C-D.

Definition 4: A *shared* dependency is a dependency presents in multiple minimal cycles. In Figure 2, the edge between A and B is shared by the two minimal cycles A-B-E and A-B-C.

B. Feedback Arcset

In graph theory, a feedback arcset is a collection of edges we should remove to obtain an acyclic graph. The *minimum* feedback arcset is the minimal collection of edges to remove to break the cycle. This approach could produce good results working on package dependencies because it does not break so much the structure. This method is not usable for two important reasons:

- It is a NP-complete problem (optimized by Kann [Kan92] to become APX-hard).
- It does not take into account the semantic of the software structure. Optimizing a graph is not equivalent to a possible realistic at the software level.

C. Cycle Visualization

1) Graph Visualization: Figure 1 shows a sample graph with five nodes and three minimal cycles. Notice that cycle A-B-C and A-B-E share a common dependency (in bold) from A to B. This shared dependency is interesting to spot since it joins two cycles and by removing it we would break those cycles.

Graph layouts offer an intuitive representation of graphs, and some handle cyclic graph better than others. On large graphs, complex layouts may reduce the clutter but this is often not simple to achieve.

2) DSM and eDSM Visualization: DSM (Dependency Structural Matrix) provides a good approach to see software dependencies [Ste81], [SGCH01], [LF05], [SGS+05], [SJSJ05] and particularly cycles [SJSJ05]. It provides a dependency-centric view possibly associated with color to perceive more rapidly some values [HBO10]. We use DSM (Dependency Structural Matrix) to see direct cycles: a direct cycle is displayed in red and the two cells representing the two dependencies of the direct cycle are symmetric along the diagonal [SJSJ05]. Seeing indirect cycles is more difficult, as the visualization is not adapted for it. The main reason for this problem is that it is difficult to read an indirect cycle in the matrix, *i.e.,* to follow the sequence of cells representing the sequence of dependencies in the cycle. The order can appear quite arbitrary as one has to jump between different columns and rows (this problem does not exist with direct cycles as there is only two cells involved, mirroring each other along the diagonal). The cycle A-B-E composed by the three dependencies A>B, B>E and E>A has been circled in Figure 3 to show the complexity of reading indirect cycles, intertwined with direct cycles.

In Figure 3, the whole matrix displays a pale blue background, indicating that A, B, C, D, and E are in the same SCC. We can see the direct cycle between C and D (in red) and in yellow the other dependencies in the SCC.

We propose eDSM [LDDB09], an improvement of DSM, which shows the relationships between classes in package dependencies. It shows all classes involved in a dependency and which types of dependency exist, providing a good

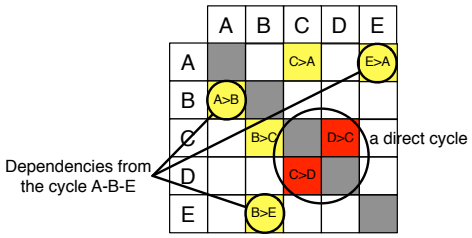

Figure 3. DSM corresponding to the graph of Figure 1. Each cell represents a dependency.

understanding of the dependency and support for breaking the dependency when necessary. While eDSM allows us to analyze direct cycles comfortably, it does not address the problem of indirect cycles left over after removal of direct cycles.

D. Lack of Solutions

In this section, we presented the problem of understanding and breaking cycles and we explained why existing approaches are not up to the task. Solving cycles in legacy systems with several packages and large SCCs is difficult. Feedback Arcset is not necessarily adapted. Node-link representations become unreadable with a large number of packages and dependencies crossing each other. DSM does not provide enough information about indirect cycles. Based on our experience, we propose to focus on shared dependencies in order to efficiently understand and break cycles. The next section shows how this heuristic is embodied and used in the CycleTable visualization.

III. CycleTable

During our experiments with eDSM, we have noted that a dependency can be part of multiple cycles. These "shared" dependencies should be highlighted because when we remove them, all involved cycles disappear. Our intuition is that the more shared a dependency is, the more likely it is unwanted in the architecture and should be removed.

We propose a visualization to help reengineers to identify dependencies involved in cycles and to highlight shared dependencies. This visualization shows all minimal cycles ordered by shared dependencies.

A. CycleTable in a Nutshell

We design CycleTable with the purpose of visualizing intertwined cycles. CycleTable is a rectangular matrix where packages are placed in rows and cycles in columns. CycleTable (i) shows each minimal cycle independently in columns; (ii) highlights shared dependencies between minimal cycles; and (iii) uses context displaying cells to provide information about dependency internals, enabling *small multiples and micro-macro reading* [Tuf97] *i.e.,* variations of the

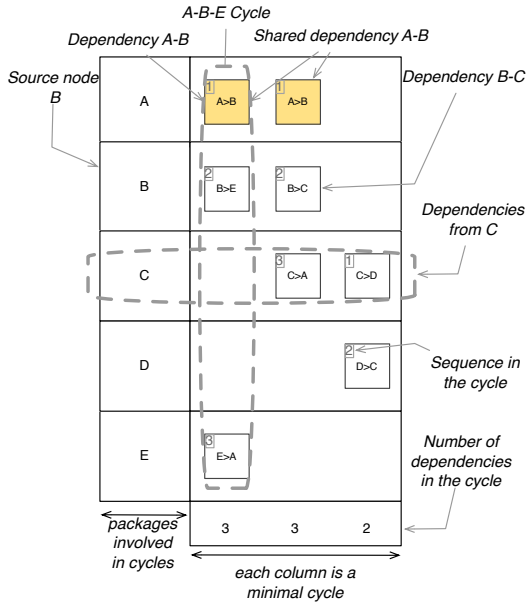

Figure 4. CycleTable for Figure 1 sample graph.

same structure to reveal information. We detail each of these points now.

B. CycleTable Layout

The CycleTable layout is presented in Figure 4. This figure shows a sample CycleTable layout for the graph in Figure 1 and Figure 2. The first column contains the name of packages involved in cycles, then each column represents one minimal cycle. A row represents all dependencies involved in cycles coming from the package. A non-empty cell, at the intersection of one row and one column, indicates that the package of this row has a dependency involved in the column cycle.

1) One cycle per column: Except for the first, each column represents a minimal cycle. In Figure 4, the first column involves packages A, B and E in a cycle (first cell represents the dependency from A to B, the second cell from B to E and the last one from E to A).

2) One package per row: Each row contains dependencies (represented as boxes) from a package. In Figure 4, the first row represents package A, with a dependency to B involved in two different minimal cycles. The second row represents package B, which depends on E and C.

3) Shared dependencies: Cells with the same background color represent the same dependency, shared by multiple cycles. In Figure 4, first row contains two boxes with a yellow background color. It represents the same dependency from A to B, involved in the two distinct cycles A-B-E and A-B-C. It is valuable information for reengineering cycles. Indeed, removing or reversing A-B can solve two cycles.

4) Size of the cycle: The last line of CycleTable displays the size of each cycle (*e.g.,* each column). This information is valuable when there are multiple cycles. A first approach could be to fix the smaller cycles because there are fewer dependencies to understand.

C. Cycle Sequence

Cycle sequence represents a relative order between dependencies. This number is sometimes necessary to retrieve the exact order of dependencies in a cycle. Let's take the example of cycle A-B-E. In Figure 4, the first relative dependency is A>B (there is the number 1 in top-left corner). The second dependency of the cycle is B>E (number 2 in top-left corner). The third and last dependency of the cycle is E>A (number 3 in top-left corner). In this particular case, it is not useful as cycle sequence follows the top-down order.

To understand the usefulness of this information, Figure 5 provides a real example. The 13^{th} column displays a cycle that cannot be read from top to bottom. As the dependencies are not in the right order, it is useful to have the sequence of the cycle. The cycle should be read from number 1 to number 5, 1: FxExtension>MsFinder, 2: MsFinder>MsWizard, 3: MsWizard>MsGene, 4:MsGene>MsCore and 5: MsCore>FxExtension.

IV. Reading a CycleTable

Figure 5 shows a sample CycleTable with 9 packages involved in 15 minimal cycles.

A. Reading a package

There are three visualization patterns for a package.

- There is one color in the row: the package has one shared dependency to another package but it is involved in multiple cycles. For example in Figure 5, the package MsCore (row 2) has one shared dependency to FxExtension, this dependency is shared by all cycles displayed in this CycleTable.
- There are white cells in the row: a white dependency is not a shared dependency. The package is involved in multiple cycles with many dependencies. In Figure 5, the package MsWizard (last row) has three different dependencies involved in three different cycles. There is no shared dependency.
- When there are multiple colors in the row, there are multiple cycles with multiple shared dependencies. It is the common visualization pattern. The goal is to look at the most present color. For example in Figure 5, the line MsMont has six cells: four cells are white and two cells are green. The two green cells are the same dependency, the white cells represents four different dependencies. MsMont has five different dependencies, involved in six cycles.

B. Reading a cycle

A column represents a cycle. Cycle length is displayed at the bottom of the table. A cycle with colored cells has shared dependencies with other cycles. For example in Figure 5, the 9^{th} cycle between FxCore, MsCore and FxExtensions has two shared dependencies (red and blue) and one non-shared dependency.

C. Reading Colors

The more cells share the same color, the more the same dependency is involved in multiple cycles. Then this dependency is interesting for cycle removing. We do not say that this dependency must be removed, but when we remove a shared dependency, all cycles involving this dependency are removed. For example, in Figure 5, if the blue dependency from MsCore to FxExtension could be removed, all presented cycles would be removed.

V. CycleTable with enriched cells

A package is a complex structure containing multiple classes in relation with other classes inside and outside the package. Showing the details at class level of a dependency from one package to another is also important to understand the dependency and assess its weight. We use a small view in each cell (named *eCell* for *enriched cell*). A first version was proposed in [LDDB09] and has been adapted to CycleTable. Figure 7 shows how each eCell provides a closed context to understand each dependency separately, yet allows one to compare the complexity of dependencies with each other. As the subject of this paper is not eCell, this section only shows the behavior that we can use in cells.

A. Overall structure of an enriched cell

An enriched cell contents displays all dependencies at class level from a source package to a target package.

An enriched cell is composed of three parts (see Figure 6):

- At the bottom is a colored frame which represents the identification of shared dependencies by a color and a number for sequence identification.
- The top row gives an overview of the strength and nature of dependencies between classes into the two involved packages. It shows the total number of dependencies (Tot) in black, inheritance dependencies (Inh) in blue, references to classes (Ref) in red, invocations (Msg) in green, and class extensions (Ext) made by the source package to the target one in gray[1].
- The two large boxes in the middle detail class dependencies going from the top box to the bottom box

[1] A class extension is a method defined in a package, for which the class is defined in a different package [BDN05]. Class extensions exist in Smalltalk, CLOS, Ruby, Python, Objective-C and C#3. They offer a convenient way to incrementally modify existing classes when subclassing is inappropriate. They support the layering of applications by grouping a package with its extensions to other packages. AspectJ inter-type declarations offer a similar mechanism.

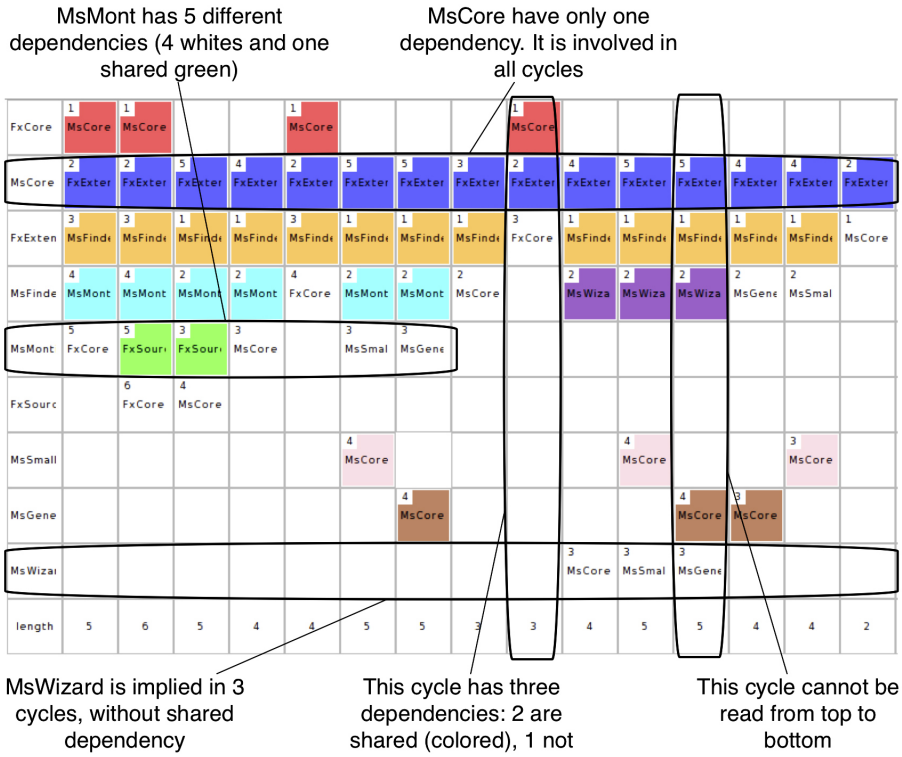

Figure 5. A subset of Moose in CycleTable with simple cells.

i.e., from the *source package* to the *target package*. Each box contains squares that represent involved classes: referencing classes in the source package and referenced classes in the target package. Dependencies between squares link each source class (in top box) to its target classes (in bottom box) (Figure 6).

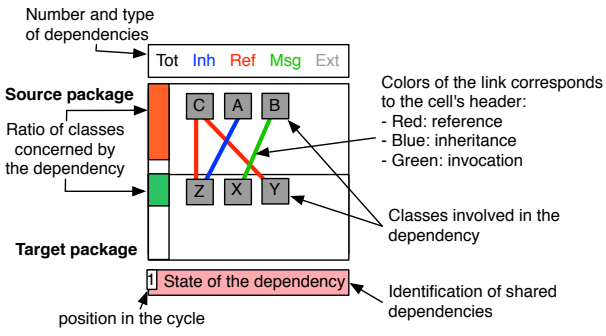

Figure 6. Enriched cell structural information.

B. Breaking Cycles with an enriched CycleTable

Figure 7 shows a CycleTable with four packages of Moose core: FxCore, MsCore, FxExtensions and MsFinder. Six minimal cycles are immediately visible. It also appears that three dependencies are each involved in multiple cycles (with red, blue, and orange frames).

An important asset of CycleTable is that it does not focus on a single solution to break cycles. It rather highlights different options as there are many ways to resolve such cycles. Only the reengineer can select what he thinks is the best solution. We now discuss how CycleTable allows one to consider solutions for solving cycles in Figure 7.

The first point to consider in CycleTable is the notion of shared dependencies, the number of cycles that are involved, and their weight. For example, the red cell linking FxCore to MsCore (first row) is in two indirect cycles and one direct cycle. It has a weight of two dependencies and involves four classes (two in each package) as well. But one dependency is an inheritance which can require some work to remove. Finally, from a semantic point of view, MsCore is at the root of many functions in the system so it seems normal to have such dependencies from FxCore.

Instead, we turn our focus to the blue cells (named A in Figure 7), linking MsCore to FxExtensions. It has a weight of five dependencies and involves two classes. From a semantic point of view, FxExtensions represents extended functionalities of the system so it seems that the dependency from MsCore is misplaced: it is just a single method referencing a class in FxExtensions. Moving the method to package FxExtensions is enough in this case to remove the

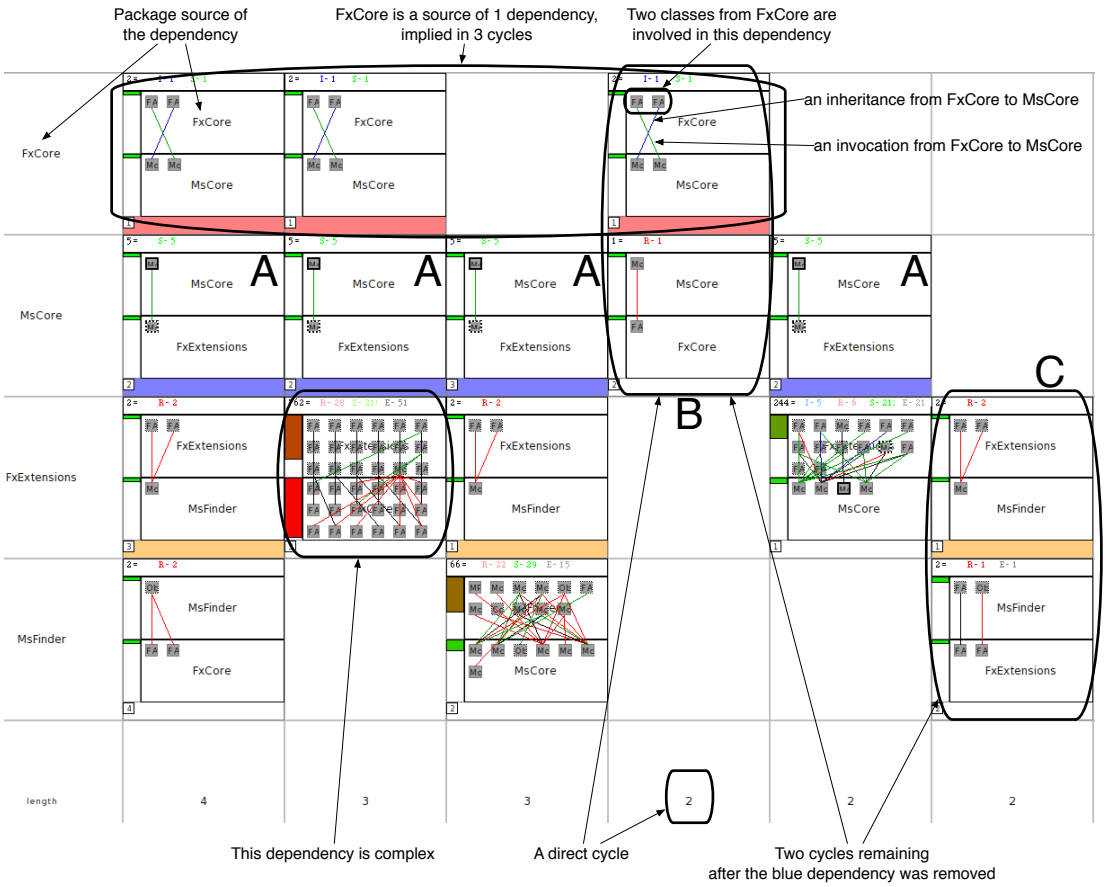

Figure 7. CycleTable with enriched cells. There are 4 packages involved in cycles: FxCore, MsCore, FxExtensions and MsFinder

dependency. This single action breaks four cycles.

Two direct cycles remain: (FxCore - MsCore) named B in Figure 7 and (FxExtensions - MsFinder) named C in Figure 7. The cycle C has a dependency shared with previously fixed cycles (yellow dependency) and is small (two internal dependencies). But the other dependency is also made of two internal dependencies. The situation is balanced. In this case the reengineer has to rely first on his knowledge of the system architecture to detect the improper dependency (FxExtensions >MsFinder). CycleTable is still useful to explore the involved classes and methods.

We assessed before that the dependency from FxCore to MsCore is acceptable. Hence, the dependency from MsCore to FxCore should be removed to resolve the cycle labeled B (Figure 7). As for the first case, a single method making a reference was misplaced in package MsCore and should become a class extension.

VI. Validation

We performed two studies to validate our approach. First, we show on a case study that unexpected dependencies in the architecture, which should be removed, often reveal themselves as shared dependencies and are given the primary focus in CycleTable. Second, we perform a comparative study of CycleTable with a normal node-link visualization to validate the efficiency of CycleTable when understanding and fixing large sets of intertwined cycles.

A. Case study on unexpected dependencies

1) Protocol: The case study was realized on the core of *Moose version 4beta4* (33 packages). The rest of Moose is not included in this case study because it does not have cycles. A developer from the Moose team evaluated all package dependencies of the system (106 dependencies), regardless of their involvement in cycles. The goal was to retrieve an objective evaluation of each dependency. The possible values that the developer can give are: the dependency is expected in the system architecture, purpose of the dependency requires deep investigation, and the dependency is unexpected and should probably be removed.

After this step, we match all shared dependencies from CycleTable against the evaluation given by the developer. We assessed two hypotheses: the probability that unexpected dependencies are often shared, and prominence of unexpected

dependencies in CycleTable, given by their positions in the matrix.

2) Results—shared dependencies as primary targets for removal investigation: Table I summarizes the results of the case study. The first three lines show some characteristics of the system: there are 14 packages involved in 42 minimal cycles, themselves including 17 different shared dependencies. Then, the assessment performed by the Moose developer returned 11 unexpected dependencies, which should be removed. Finally, we perform the intersection between unexpected and shared dependencies: 9 out of the 11 unexpected dependencies are also shared by various cycles. The two other unexpected but not shared dependencies are actually independent direct cycles *i.e.,* they are direct cycles forming each one SCC, with no intertwined cycles. These two dependencies are not critical in the system architecture.

The 11 unexpected dependencies retrieved by the developer cover the 42 minimal cycles: in other words, fixing those 11 dependencies would break all cycles. It is remarkable that fixing the 9 shared dependencies actually break 40 out of 42 minimal cycles (the two remaining cycles being the independent direct cycles). This case study shows that i) unexpected dependencies are often shared dependencies, and that ii) removing shared dependencies can break multiple cycles with minimal effort, as only a handful of dependencies need to be assessed.

Characteristics	Moose
number of packages	33
number of packages in cycles	14
number of dependencies	106
number of minimal cycles	42
number of shared dependencies	17
number of unexpected dependencies	11
unexpected ∩ shared	9
cycles coverage by unexpected ∩ shared	40 / 42

Table I
RESULTS OF SHARED DEPENDENCY VALIDATION.

3) Results—prominence of unexpected dependencies in CycleTable: CycleTable uses a heuristic to order packages and cycles in the matrix. This heuristic tries to place cycles sharing common dependencies next to each other. In this study, we show that the ordering given by the heuristic effectively also puts forward unexpected dependencies, given that they are often shared. Starting with the set of unexpected dependencies retrieved by the developer, we looked up the position of the source package in CycleTable. This position corresponds to the row where the dependency is displayed.

Table II shows that 9 out of 11 unexpected dependencies (80%) appear in the first three lines (3 out of 15 packages, 20%). Thus issues with cyclic dependencies relate mostly to three packages. This result shows that just by focusing on the first part of the visualization, a great deal of work can be done in breaking cycles.

Unexpected dependency	Position in CycleTable (line number)
Famix-Core » Famix-Implementation	**1**
Moose-Core » Famix-Core	**2**
Moose-Core » Moose-SmalltalkImporter	**2**
Moose-Core » Famix-Extensions	**2**
Moose-Core » Moose-GenericImporter	**2**
Moose-Core » Famix-Implementation	**2**
Famix-Extension » Famix-Smalltalk	**3**
Famix-Extensions » Moose-Finder	**3**
Famix-Extension » Famix-Java	**3**
Fame » Moose-Core	9
Moose-Wizard » Moose-Finder	10

Table II
RESULTS OF UNEXPECTED DEPENDENCY POSITION.

B. Comparative Study with Node-link Representation

1) Protocol: In this comparative study, we validate CycleTable as a useful visualization to understand and break a large set of cycles intertwined together. The precise goal of the study was to validate the effectiveness of CycleTable layout in matrix, compared to a common node-link layout. We measure the time taken by participant to reason about cycles and the quality of their answer.

The protocol is the following: first the participant is given a tutorial about the task and the tool with a small example, questions and correct answers to train himself. Second he performs the same questions on the real case study. For the case study, we use a subset of Moose (the 14 packages in cycles, see Table I). Since we focus on assessing the tool, we replace all package names by arbitrary letters from A to O and we do not use enriched cell (Figure 9). Hence, participants could not use prior Moose background (some had already work as developers in Moose) or package names to guide their intuition. The assessment of multiple intertwined cycles is impractical when one uses a single node-link representation showing the full SCC (as shown in Figure 8).

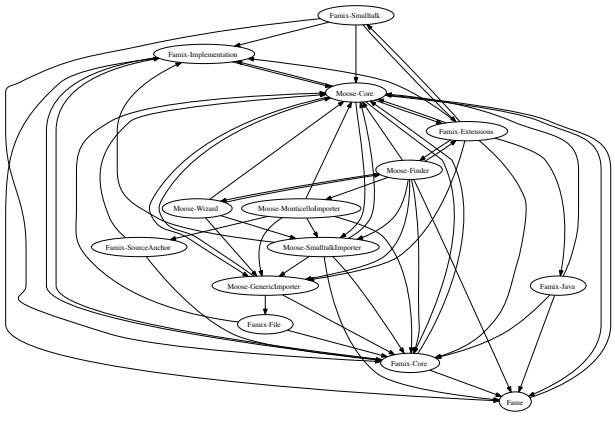

Figure 8. Node-Link representation of the Moose 14 packages in cycle.

Instead, we choose to display a series of node-link representations, one for each minimal cycle built with dotty/GraphViz [GN00]. This allows us to map the same data in CycleTable and node-link. In particular, a shared dependency was also displayed with the same color across multiple node-links.

One group of seven participants answers questions using CycleTable. The other group of six participants answers questions using the node-link visualization.

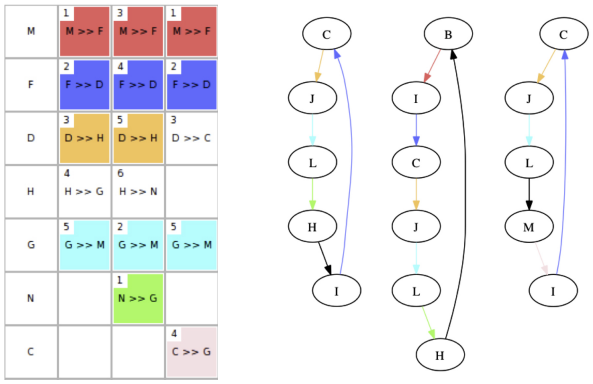

Figure 9. Sample of CycleTable (left) and node-link visualization(right) used in the study.

2) List of questions: Here are the eight questions that the users have to answer. We also give the rating of the answer, based on the distance of the answer to the correct one. A 0 rating indicates a good answer.

- Q1: Give 2 packages that are in a direct cycle (cycle between two packages). *Rating:* 0 when the answer represents a direct cycle, 1 else.
- Q2: Give a minimal cycle involving N and O (enumerate packages in order). *Rating:* 0 when the answer is {O, G, N}, +1 for each false value.
- Q3: How many minimal cycles go through package F? *Rating:* Computed as the difference between the answer and 16, the correct answer.
- Q4: How many shared dependencies exit package F? *Rating:* Computed as the difference between the answer and 2, the correct answer.
- Q5: How many dependencies should be removed to break all cycles involving package F? *Rating:* Computed as the difference between the answer and 8, the correct answer.
- Q6: What is the biggest shared dependency in the system? *Rating:* 0 when the answer is {G, M}, 1 else.
- Q7: How many minimal cycles are broken by removing the biggest shared dependency? *Rating:* Computed as the difference between the answer and 24, the correct answer.
- Q8: Give the minimum number of edges to remove in order to break all cycles in the system. *Rating:* Computed as the difference between the answer and 10, the correct answer.

3) Results: 13 participants performed the study, from license's degree students to experienced researchers with various programming skills and experience in visualizations. We distinguish three parts in the results (Figure 10).

The first part relates to questions Q1 and Q2. For these two easy questions, the user should identify cycles. Results shows that it is faster to identify a cycle with a node-link visualization. We can consider that it is still more intuitive than CycleTable.

The second part relates to questions 3 to 7, where the user should recognize shared dependencies or packages involved in multiple cycles. Here, CycleTable performs better and faster. Q7 appears as an exception: actually participants in both groups confused two very similar colors, which is a mistake on our part for the choice of colors. These questions validate the design of CycleTable compared to node-link, for the purpose of reasoning about shared dependencies.

Finally question 8 evaluates the capacity to assess the full complexity of the graph. It builds upon the preparatory work made by answering the previous questions as one needs to assess a minimal set of dependencies, mostly based on the impact of removing shared dependencies. Results show it takes in average more than 90 seconds with CycleTable and more than 3 minutes with node-link visualization. While both groups gave similar answers, it highlights the ease to read CycleTable for this task.

C. Threats to validity

Rating.: We compute a rating based on the distance to the expected answer. We can see that CycleTable provides better answers than a node-link visualization. But false-answers are due to a visualization without software meaning for participants. In the case of real reengineering session, results could be different. It is a part of future work.

Removable Dependency.: We suppose for CycleTable that a critical dependency is often shared. In our case study, results show that this hypothesis is right. But we should do more experiments to confirm it.

Smalltalk Software.: Smalltalk software applications have specific features as class extension, which makes it easier to modularize software, but which makes it also easier to make cycles. A work is in progress to analyze Java software.

D. Conclusion of the Study

We create this visualization to assess cycles at package level. To analyze the usefulness of the visualization, we do not have other visualization tool to compare. We build a node-link visualization which shows shared dependencies. The benefit of node-link visualization is that there is no learning time.

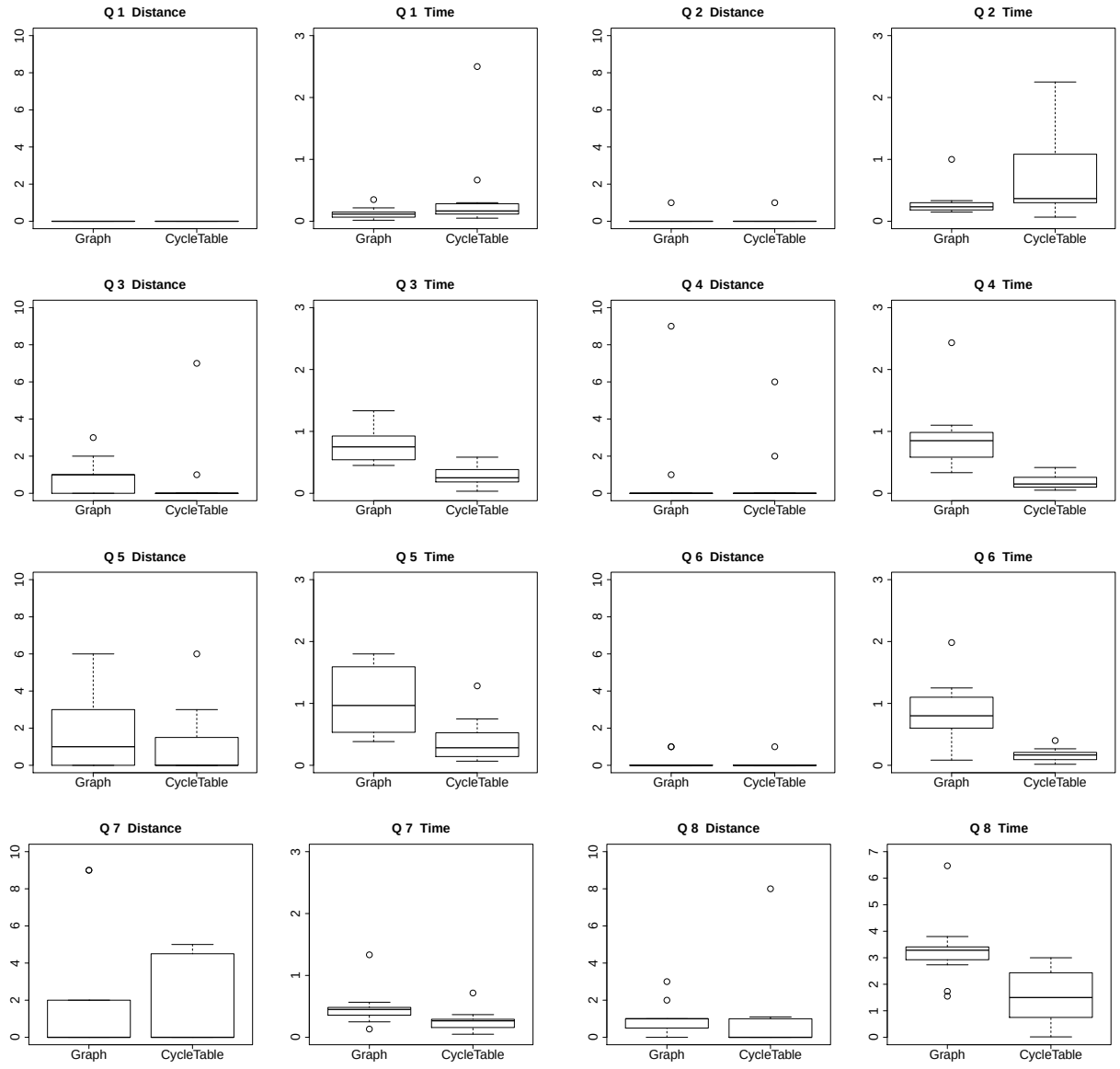

Figure 10. Boxplots showing distance to expected answer in absolute and time in minutes for questions 1 to 8. Graph shown on left and CycleTable on right for each question.

The study shows that CycleTable is efficient to detect and to help reengineer breaking cycles between packages. There are still some limits that we would like to overcome, with the goal to make CycleTable more effective for reengineers.

The order of cycles in the matrix is based on the similarity they share with each other, based on their common shared dependencies. The order of packages follows cycle sequences as soon as cycles are inserted in the matrix. This heuristic gives good result in the first rows and columns of the visualization. Then it becomes difficult to arrange cycles and packages so that a shared dependency forms a unique line of color in its row.

When there are cycles without shared dependencies, CycleTable shows cycles separately but without colors. Such systems are actually simple to understand. In this case the use of other visualization such as node-link or DSM could be better.

VII. Related work

Node-link visualization.: Often node-link Visualizations are used to show dependencies among software entities. Several tools such as dotty/GraphViz, Walrus or Guess can be used. Using node-link visualization is intuitive and has a short learning curve. One problem with node-link visualization is finding a layout scaling on large sets of

nodes and dependencies: such a layout needs to preserve the readability of nodes, the ease of navigation following dependencies, and to minimize dependency crossing. Even then, cycle identification is not trivial.

Package Blueprint.: It shows how one package uses and is used by other packages [DPS+07]. It provides a fine-grained view. However, package blueprint lacks (1) the identification of cycles at system level and (2) the detailed focus on classes actually involved in the cycles.

Dependency Structural Matrix.: Contrary to node-link, a DSM visualization preserves the same structure whatever the data size is. This enables the user to dive fast into the representation using the normal process. SCCs can be identified by colored cells. Moreover, eDSM [LDDB09] displays fine-grained information about dependencies between packages. Classes in source package as well as in target package are shown in the cells of the DSM.

Dependence Clusters.: Brinkley and Harman proposed two visualizations for assessing program dependencies, both from a qualitative and quantitative point of view [BH04]. They identify global variables and formal parameters in software source-code. Subsequently, they visualize the dependencies. Additionally, the MSG visualization [BH05] helps finding *dependence clusters* and locating avoidable dependencies. Some aspects of their work are similar to ours. Granularity and the methodology employed differ: they operate on source-code and use slicing method, while we focused on coarse-grained entities and use model analysis.

VIII. Conclusion

This paper proposes CycleTable, a visualization showing cycles between packages in order to break cyclic dependencies. A fundamental heuristic of CycleTable is the focus on "shared dependencies", which can impact many cycles at once by their removal. The visualization can be completed with eCell which has been integrated in eDSM [LDDB09].

We validate the heuristic of "shared dependencies" in a case study and the efficiency of CycleTable over a node-link visualization in a comparative study. We plan to work on (i) pursuing the validation of "shared dependencies", (ii) applying CycleTable on several software applications, in particular in other languages, and (iii) making an approach with both node-link visualization and CycleTable as they are complementary.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013'.

References

[BDN05] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 11(10):1613–1644, November 2005.

[BH04] David Binkley and Mark Harman. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Trans. Softw. Eng.*, 30(11):715–735, 2004.

[BH05] David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In *ICSM '05*, pages 177–186, Washington, DC, USA, 2005. IEEE Computer Society.

[DPS+07] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham 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.

[GN00] Gansner and North. An open graph visualization system and its applications to software engineering. *Software Practice Experience.*, 30(11):1203–1233, 2000.

[HBO10] Jeffrey Heer, Michael Bostock, and Vadim Ogievetsky. A tour through the visualization zoo. *Queue*, 8(5):20–30, 2010.

[Kan92] Viggo Kann. *On the Approximability of NP-complete Optimization Problems*. PhD thesis, Royal Institute of Technology Stockholm, 1992.

[LDDB09] J. Laval, S. Denier, S. Ducasse, and A. Bergel. Identifying cycle causes with enriched dependency structural matrix. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Lille, France, 2009.

[LF05] Antónia Lopes and José Luiz 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.

[Mar00] Robert C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.

[SGCH01] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ESEC/FSE 2001*, 2001.

[SGS+05] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *Proceedings of the ESEC/SIGSOFT FSE 2005*, pages 166–175, 2005.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[Ste81] 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.

[Tuf97] Edward R. Tufte. *Visual Explanations*. Graphics Press, 1997.