Matrice de dépendances enrichie

Jannik Laval, Alexandre Bergel, Stéphane Ducasse, Romain Piers

RMoD Team, INRIA - Lille Nord Europe - USTL - CNRS UMR 8022, Lille, France
firstname.lastname@inria.fr

Résumé. Les matrices de dépendance (DSM - Dependency Structure Matrix), développées dans le cadre de l'optimisation de processus, ont fait leurs preuves pour identifier les dépendances logicielles entre des packages ou des sous-systèmes. Il existe plusieurs algorithmes pour structurer une matrice de façon à ce qu'elle reflète l'architecture des éléments analysés et mette en évidence des cycles entre les sous-systèmes. Cependant, les implémentations de matrices de dépendance existantes manquent d'informations importantes pour apporter une réelle aide au travail de réingénierie. Par exemple, le poids des relations qui posent problème ainsi que leur type ne sont pas clairement présentés. Ou encore, des cycles indépendants sont fusionnés. Il est également difficile d'obtenir une visualisation centrée sur un package. Dans ce papier, nous améliorons les matrices de dépendance en ajoutant des informations sur (i) le type de références, (ii) le nombre d'entités référençantes, (iii) le nombre d'entités référencées. Nous distinguons également les cycles indépendants. Ce travail a été implémenté dans l'environnement de réingénierie open-source *Moose*. Il a été appliqué à des études de cas complexes comme le framework *Morphic UI* contenu dans les environnements Smalltalk open-source *Squeak* et *Pharo*. Les résultats obtenus ont été appliqués dans l'environnement de programmation *Pharo* et ont mené à des améliorations.

1 Introduction

Comprendre la structure de grosses applications est un défi mais aussi une tâche importante pour la maintenance (Demeyer et al., 2002). Plusieurs approches présentent des informations à propos des packages et de leurs relations, par la visualisation d'artefacts logiciels, de leur structure, de leur évolution ou encore de métriques. Les métriques logicielles peuvent être difficiles à comprendre puisqu'elles dépendent fortement de l'application analysée. Distribution Map (Ducasse et al., 2006) résoud ce problème en montrant comment les propriétés se répartissent dans une application. Langelier et al. (2005) caractérisent l'évolution des packages et de leurs métriques. Package Surface Blueprint (Ducasse et al., 2007) révèle la structure interne d'un package et les relations avec les autres packages — le concept de surface représente les relations entre le package analysé et les packages auxquels il accède. Dong et Godfrey (2007) ont travaillé sur la présentation de systèmes par des graphes de dépendances objets de haut niveau pour aider à la compréhension des systèmes au niveau de leurs structures de packages.

Les matrices de dépendance (DSM - Dependency Structure Matrix) sont une solution éprouvée pour révéler les problèmes de cycles dans une structure (Sangal et al., 2005). Développées à l'origine pour l'optimisation des processus, elles mettent en évidence des problèmes de dépendances entre tâches. La technique a été appliquée avec succès pour l'identification de dépendances applicatives (Steward (1981); Sullivan et al. (2001); Lopes et Fiadeiro (2005)). MacCormack et al. (2006) ont appliqué les matrices de dépendance pour analyser le niveau de modularité de l'architecture de Mozilla et Linux.

Cependant, appliquéesà la réingénierie logicielle, elles n'offre pas une solution optimale : elles ne fournissent pas suffisamment d'informations exploitables. Les DSMs utilisées de nos jours permettent juste de faire un état des lieux, sans réellement offrir une aide à la re-ingéniérie. Par exemple, les implémentations actuelles de DSM ne fournissent pas d'informations à propos des types de liens entre les entités. Certains algorithmes (comme l'élévation à la puissance des matrices d'adjacences) n'affichent pas tous les cycles indépendamment les uns des autres. Il est également difficile de détecter les cycles propre à une entité précise. De plus, les matrices de dépendance ne prennent pas en compte les extensions de classes, construction élémentaire dans les languages dynamiquement typés[1].

Notre contribution est double : d'une part, nous identifions les faiblesses des matrices de dépendance traditionnelles (Section 2), d'autre part, nous répondons à ces faiblesses (Section 3). Nous appliquons notre outil sur le package *Morphic UI* afin de le remodulariser et l'intégrer à *Pharo*, une nouvelle version de Squeak. Nous proposons des matrices de dépendances enrichies (EDSM). Nous enrichissons les cellules des matrices de dépendance avec des informations contextuelles en affichant (i) les types de références existantes (héritage, accès aux classes, ...), (ii) la proportion d'entités (classes/méthodes) origines, (iii) la proportion d'entités cibles. Nous distinguons également les cycles indépendants et utilisons des informations colorimétriques pour cela. Finalement nous offrons une vision par niveau des cycles entre les entités.

Le document est organisé de la façon suivante : la Section 2 présente les DSM et leurs limites actuelles. La Section 3 présente notre solution. La Section 4 montre un exemple de mise en œuvre sur l'application *Morphic UI*. La Section 5 présente les limites de notre solution. La Section 6 conclue le document.

2 Limites des matrices de dépendance

Les matrices de dépendances (DSM) sont efficaces pour détecter des cycles entre composants logiciels. Bien que ces résultats soient pertinents pour vérifier l'indépendance de différents composants logiciels (Sangal et al., 2005), les DSM sont moins utiles pour la re-ingénierie.

Nous avons identifié un certain nombre de limites aux DSM : la fusion de certains cycles indépendants (Section 2.1), le manque de précision de la visualisation (Section 2.2), l'absence d'information de cycles pour une entité précise (Section 2.3) et le manque de support des extensions de classes (Section 2.4).

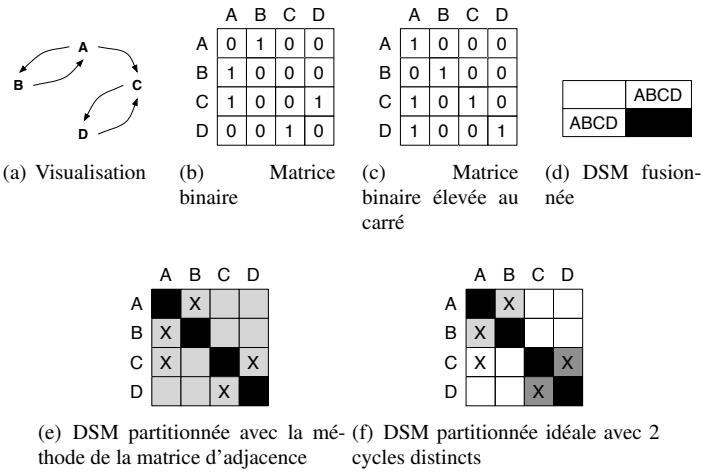
[1] Une extension de classe est le fait qu'une méthode peut être définie dans un autre package que celui de sa classe. Les langage Objective-C, Smalltalk, C# 3.0 offrent différents degrés d'extensions de classes.

2.1 Cycles flous avec la méthode d'élévation à la puissance

Un des algorithmes utilisés pour le calcul des cycles est basé sur l'élévation à la puissance de la matrice d'adjacence. Le principe de cette approche est d'élever une DSM binaire à sa énième puissance pour indiquer quels éléments peuvent emprunter un chemin revenant sur eux-mêmes en n étapes et donc constituer un cycle (Yassine et al., 1999). Cet algorithme est limité car les cycles de même longueur ne sont pas différenciés.

Le formalisme utilisé pour la lecture des DSM est le suivant : les éléments en tête de colonne font appel aux éléments en tête de ligne. Par exemple, sur la Figure 1(b), A fait appel à B et à C, B fait appel à A, C fait appel à D et D fait appel à C.

Sur la Figure 1(a), les éléments A et B constituent un cycle direct et C et D en constituent un autre. Mais si nous élevons la DSM binaire (Figure 1(b)) au carré, une valeur différente de zéro apparaît dans la diagonale de chacun des éléments (Figure 1(c)). Ces valeurs signifient que chacun des éléments A, B, C et D est impliqué dans au moins un cycle direct mais ces valeurs ne montrent pas la répartition des éléments dans les cycles directs. Ainsi l'algorithme fusionne ces 4 éléments (Figure 1(d)), ce qui signifie que ces éléments apparaissent comme un seul cycle (Figure 1(e)) – notons que la zone grise représente les cycles. Ainsi, la matrice de partitionnement fournit une information erronée en indiquant un cycle unique (la zone grise dans la Figure 1(e)) alors que la matrice devrait montrer 2 cycles directs comme dans la Figure 1(f).



(a) Visualisation (b) Matrice binaire (c) Matrice binaire élevée au carré (d) DSM fusionnée

(e) DSM partitionnée avec la méthode de la matrice d'adjacence (f) DSM partitionnée idéale avec 2 cycles distincts

FIG. 1 – *Limite de la méthode de puissance de matrice d'adjacence*

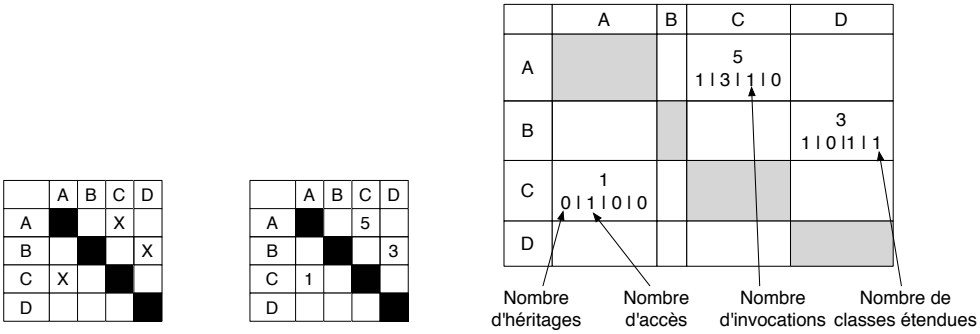
L'élévation à la puissance de la matrice d'adjacence ne permet pas de déterminer précisément les cycles différents. Cependant, nous utilisons cet algorithme combiné à une méthode de recherche de chemin afin d'identifier les différents cycles de même niveau.

2.2 Manque de précision de la visualisation

Une DSM traditionnelle offre un aperçu général mais sans information précise de la situation qu'elle décrit. Nous identifions deux sortes de faiblesses : le manque d'information sur les

causes des dépendances d'une part, et sur leur importance d'autre part.

Manque d'information sur les causes. Les dépendances peuvent avoir plusieurs causes : des accès aux classes, des extensions de classes (voir la Section 2.3), des relations d'héritage et des invocations de méthodes. Les coûts d'élimination d'un cycle varient en fonction du type de lien choisi pour suppression : changer une référence directe à une classe est souvent plus simple que de changer une relation d'héritage. C'est pourquoi il n'est pas suffisant d'indiquer les dépendances dans une DSM avec un simple marqueur *X* (Figure 2(a)) ou même avec un nombre représentant le nombre de dépendances qui existent (Figure 2(b)). Nous pensons qu'indiquer au moins un nombre pour chaque type de dépendance (Figure 2(c)) donne une indication plus précise et aide ainsi à une meilleure compréhension de la situation.



(a) DSM avec des marqueurs

(b) DSM avec des nombres

(c) DSM avec les types de dépendance

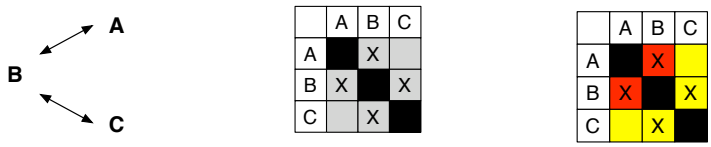
FIG. 2 – *Exemple de références dans une DSM*

Manque d'information sur les impacts. De plus, certaines informations importantes ne sont pas données et nous n'avons aucune idée des problèmes potentiels. Par exemple, le fait qu'un package ait 72 références sur un autre package (package MorphicExtra-Demo sur Morphic-Kernel présenté dans la Figure 12) est une information importante. Mais de telles références peuvent être faites par un grand nombre de classes ou bien un petit groupe et ces 72 références peuvent faire référence à un petit sous-ensemble ou à un grand nombre de classes. La même remarque peut être faite pour les méthodes. Ces informations supplémentaires sont pertinentes et importantes. Par exemple pour le package MorphicExtra-Demo, 6 classes et 45 méthodes sont concernées et elles appellent 2 classes et 40 méthodes du package Morphic-Kernel. En plus, avoir accès à ces informations sans avoir à regarder chaque classe permet de gagner du temps.

2.3 Manque de précision sur les cycles d'une entité

Les cycles sont vus dans le contexte du système complet. Il est difficile dans une DSM standard de comprendre le cycle (et pas son niveau) dans lequel un package donné est impliqué. En particulier, quand des cycles de même niveau sont fusionnés, nous obtenons des informations

imprécises. Par exemple, considérons un package A impliqué dans un cycle direct avec un package B. Ce package B est impliqué dans un cycle direct avec un package C (Figure 3(b)). Nous avons donc A dans un cycle qui inclut également C mais la longueur du cycle entre A et C n'est pas le même que la longueur du cycle entre le package A et le package B. Voir ces différences de longueurs de cycle (Figure 3(c)) est une information importante parce que les méthodes utilisées pour casser des cycles de différentes longueurs ne seront pas les mêmes. Dans la Figure 3(c) les cycles incluant l'entité A sont montrés : les cellules rouges montrent les cycles directs avec A, alors que les cellules jaunes montrent les cycles indirects.



(a) Visualisation (b) Cycles d'un système (c) Cycles d'une entité

FIG. 3 – *L'importance des cycles appliqués à une entité précise*

2.4 Extensions de classes non prises en compte

Une extension de classe est une méthode qui est définie dans un package différent du package où est définie la classe (Bergel et al., 2005). Les extensions de classes existent en Smalltalk, CLOS, Objective-C, Ruby et maintenant en C#. Elles offrent une manière adéquate de modifier incrémentalement les classes existantes quand la construction de sous-classes est inappropriée (Figure 4).

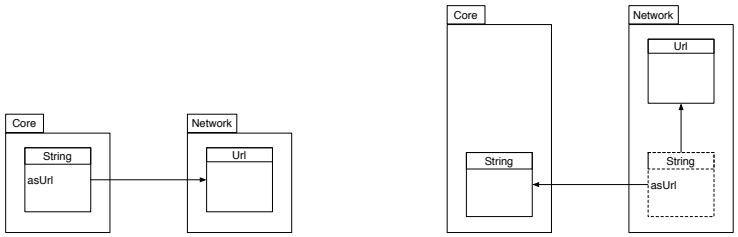
La construction de sous-classes peut être inappropriée lorsque l'on ne peut pas mettre à jour les clients d'une classe pour faire référence à une nouvelle sous-classe ou que le code source de la classe est inaccessible. Ainsi, l'extension de classe offre une bonne solution au dilemme qui apparaît lorsque l'on veut modifier ou étendre le comportement d'une classe existante sans modifier le code source de la classe en question.

Dans la Figure 4, plutôt que de créer une méthode dans le package *Core* auquel on n'a pas forcément accès, on étend la classe *String* dans le package *Network* avec la méthode asUrl. Le lien entre les deux packages est par conséquent inversé.

Cette dernière caractéristique est particulièrement importante dans l'analyse des cycles car elle offre la possibilité de supprimer un cycle en inversant la dépendance, sans pour autant devoir casser le lien qui existe entre les deux packages.

3 Une DSM enrichie

Notre approche définit une matrice enrichie (EDSM). Nous prenons en compte les limites décrites précédemment et ajoutons plusieurs fonctionnalités qui ne sont pas offertes par des logiciels de DSM industriels comme Lattix (Sangal et al., 2005) : (i) des informations enrichies dans les cellules (Section 3.1), (ii) une différenciation des cycles indépendants (Section 3.2), et (iii) le focus sur un package avec coloration des niveaux de cycle (Section 3.3).



(a) Dépendance sans extension de classe (b) Dépendance avec extension de classe

FIG. 4 – *Principe d'extension de classe*

L'outil est implémenté dans l'environnement de réingénierie *Moose*. Ainsi, l'outil profite d'une indépendance vis à vis des langages et travaille sur un modèle basé sur *FAMIX*.

3.1 Cellules enrichies

EDSM enrichit les informations contenues dans une cellule (Figure 5). Ces nouvelles informations contextuelles montrent les éléments suivants :

- *Force de la dépendance.*

  Sur la première ligne, nous montrons le nombre de références du package source vers le package cible. Ce nombre nous donne la force du lien qui existe entre ces packages. C'est une information brute telle que les DSM la présentent.

- *Type de dépendance.*

  La deuxième ligne montre les types de référence : Héritage (H), Accès (A), Invocation (I) et Extension de classe (E), et combien de références existent pour chaque type. Ces nombres nous donnent une information plus raffinée des relations entre les packages.

- *Source et diffusion de la dépendance.*

  Enfin, nous montrons comment les références sont distribuées dans les deux packages source et destination. La troisième ligne montre le nombre de classes de chaque package. Cette information indique la taille des packages impliqués dans la dépendance. La quatrième ligne indique le nombre de classes et de méthodes qui sont actuellement référençantes et référencées pour le package faisant la référence et le package référencé. La dernière ligne exprime la même idée mais avec des pourcentages. Cette information est importante parce qu'elle montre deux choses : le taux de classes sources des références et le taux de diffusion de ses références dans le package cible. Nous pouvons donc obtenir une idée de la diffusion et des flux entre les deux packages. Par exemple, si le taux de classes d'un package source des dépendances est faible, nous orientons l'effort de réingénierie pour enlever ces dépendances sur la source, alors que si ce taux est élevé, nous orienterions plutôt l'effort sur le package cible.

Ces informations contextuelles enrichies permettent d'identifier plus facilement la situation en analysant rapidement la matrice. Il est difficile d'afficher l'ensemble de la matrice à l'écran

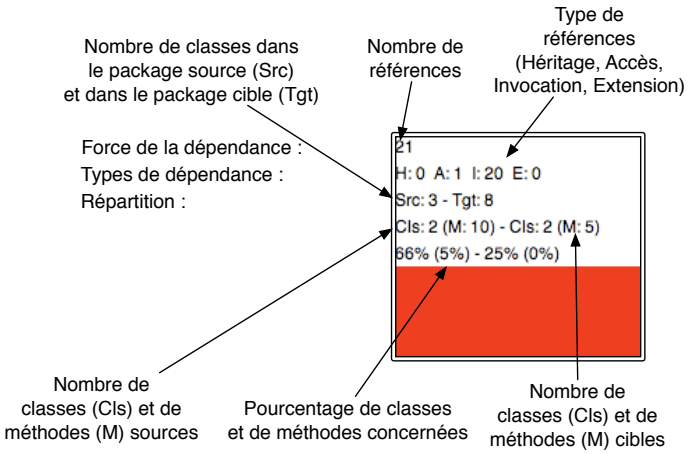


FIG. 5 – *Cellule enrichie de DSM*

en conservant la lisibilité du texte, ainsi nous recourons au zoom. En zoom arrière, la vue permet toujours d'avoir toute la matrice affichée à l'écran.

Exemple. La Figure 5 montre un exemple de cellule enrichie. Il y a 21 liens dont 1 accès (A) et 20 invocations (I), il y a 2 classes (10 méthodes) sources et 2 classes (5 méthodes) cibles. Cela représente 66% des classes (5% des méthodes) du package source et 25% des classes (0% des méthodes : le chiffre 0 apparaît en raison d'un arrondi, cela signifie que le package cible a un nombre important de méthodes) du package cible.

3.2 Meilleure détection des cycles

Dans le contexte des DSM, une alternative à l'algorithme d'élévation à la puissance de la matrice d'adjacence est un algorithme de recherche de chemin (Gebala et al., 1991). Cet algorithme nous permet de détecter séparément tous les cycles indépendants. Ainsi, comme exprimé dans la Section 2.1, deux packages, A et B, peuvent être en cycle direct et les packages C et D dans un autre cycle direct. Avec l'algorithme de recherche de chemin, ces deux cycles sont clairement identifiés. A partir de ce constat, notre approche améliore la matrice traditionnelle en proposant : la distinction des cycles , l'identification d'emboîtement des cycles , et des indications pour aider à la réingénierie des cycles.

Distinctions des cycles. Notre approche distingue les cycles indépendants. Elle est basée sur un algorithme de recherche de chemin. Avec cette méthode, deux cycles indépendants sont détectés séparément et peuvent être ainsi isolés l'un de l'autre dans la DSM (Figure 6).

Emboîtement des cycles. Nous avons ajouté des informations colorimétriques dans les cellules de la DSM pour donner des informations à propos des cycles. Ainsi, comme montré dans

Matrice de dépendances enrichie

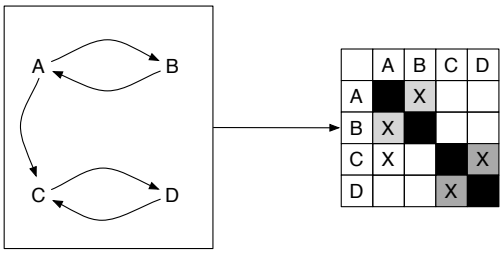
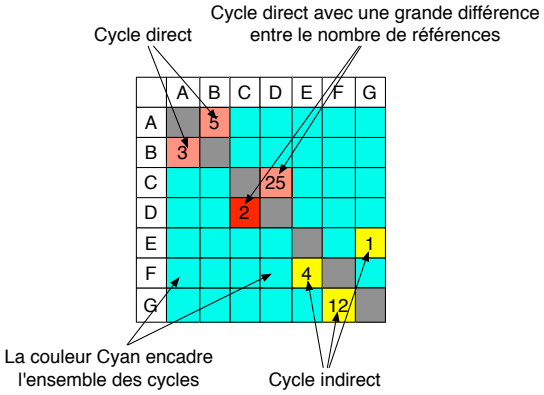


FIG. 6 – *Distinction des cycles indépendants en teinte de gris différente*

la Figure 7(a), après un partitionnement, les cellules de la DSM impliquées dans un cycle ont une couleur rouge ou jaune. La couleur rouge signifie que les deux packages concernés sont impliqués dans un cycle direct alors que la couleur jaune signifie qu'ils sont impliqués dans un cycle indirect. La surface bleu pâle entoure l'ensemble des packages qui sont en cycle (comme montré dans la Figure 8), alors que les lignes et les colonnes blanches représentent les packages sans cycle.

Indication pour la réingénierie des cycles. Dans le cas de cycles directs, s'il y a une grande différence entre le nombre de références de chacun des deux packages impliqués (nous utilisons un rapport de 3 comme seuil), la cellule qui contient le moins de références a une couleur rouge vif (Figure 7(a)). Cette information permet à l'utilisateur de centrer son attention sur les références qui peuvent être résolues a priori. Comme montré dans la Figure 8, une telle indication est vraiment utile pour repérer les packages présentant des cycles.



(a) Représentation de cycles

FIG. 7 – *Coloration des cycles directs et indirects et indication de problèmes.*

La Figure 7(a) illustre comment interpréter les cellules colorées. Elle montre qu'il y a deux cycles directs : entre A et B et entre C et D, car les cellules sont rouges. Les cycles sont

distincts car il n'y a pas de cellules rouges au croisement des lignes A ou B et des colonnes C ou D. Nous voyons que D peut être la source du problème pour le cycle avec C car il fait deux références sur C alors que C fait 25 références sur D. Puis il y a un cycle indirect entre E, F et G car il y a des cellules jaunes entre ces trois packages.

Exemple. Nous avons appliqué notre approche au framework Morphic de Squeak. Notons que Morphic n'a jamais été créé de façon modulaire. Par conséquent il présente un nombre important de dépendances cycliques. Nous observons dans la Figure 8 les packages impliqués dans des cycles (représenté par l'espace bleu pâle). Nous détaillons cet exemple plus loin. Notez que nous pouvons concentrer notre attention sur les cellules rouge vif qui indiquent les dépendances pouvant probablement disparaître pour éliminer les cycles directs.

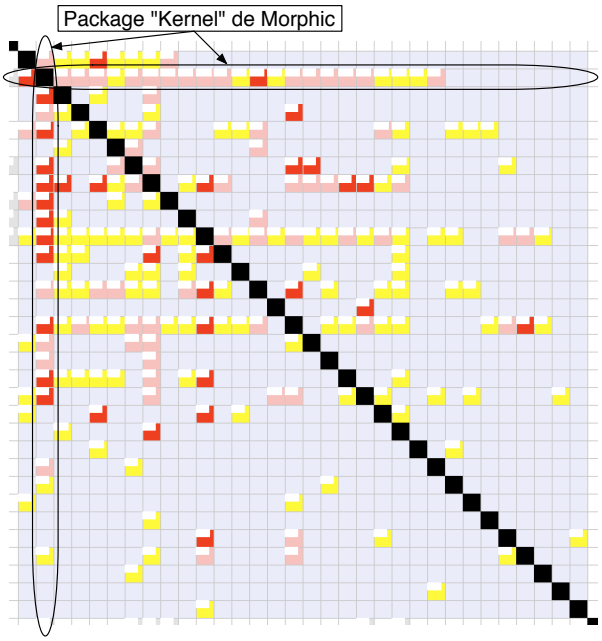


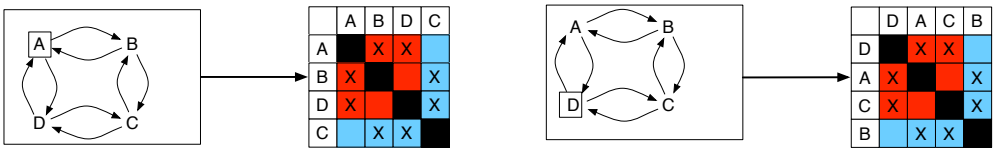
FIG. 8 – *Exemple de détection de cycle.*

3.3 Visualisation focalisée sur un package précis

Nous avons vu dans la section précédente que notre approche isole les cycles indépendants et leur ajoute des informations grâce à une coloration des cellules pour distinguer les cycles directs et indirects. Cependant, s'il est facile de voir les différents cycles qui existent à un niveau donné quand il y a peu de packages, cela devient plus difficile avec un plus grand nombre de packages. Ainsi, nous ne pouvons pas facilement connaître la longueur d'un cycle entre deux packages. En effet, ce problème apparaît particulièrement lorsqu'un élément appartient à plusieurs cycles qui n'ont pas la même longueur. Il est difficile de montrer la longueur de

tous les cycles auxquels appartient cet élément. De plus, lors d'une phase de remodularisation, l'ingénieur doit souvent porter son attention sur un package bien particulier.

Pour résoudre ce problème, nous avons ajouté la notion de package cible. Ainsi, la longueur d'un cycle entre un élément et un autre est relative à l'élément ciblé. Par exemple, dans la Figure 9, nous pouvons voir que l'élément ciblé est A, les éléments B et D sont invoqués dans un cycle direct avec A, et l'élément C est en cycle indirect avec A (Figure 9(a)). Mais si nous prenons comme élément cible le package D, les éléments A et C sont impliqués dans un cycle direct avec D et l'élément B qui est en cycle indirect avec D (Figure 9(b)).



(a) Situation avec l'élément ciblé A (b) Situation avec l'élément ciblé D

FIG. 9 – *Visualisation des différents cycles relatifs aux éléments ciblés*

3.4 Niveaux de cycle

Pour aider la visualisation des différents cycles impliquant le package cible, nous introduisons une couleur différente pour chaque niveau de cycle. En effet, comme dans la Figure 10, les éléments impliqués dans le premier niveau de cycle avec l'élément ciblé (c-à-d les éléments qui sont en cycle avec l'élément ciblé et dont la longueur du cycle est le plus court) sont de couleur rouge. Ce processus est répété pour chaque élément qui est en cycle avec l'élément ciblé : les éléments qui appartiennent au même niveau de cycle sont de la même couleur. Grâce à ces informations colorimétriques, nous pouvons d'une part facilement voir si un élément est dans un cycle avec le package cible, et d'autre part nous pouvons compter le nombre d'éléments dans chaque niveau de cycle.

4 Expériences

Pour expérimenter et valider notre approche, nous l'avons appliquée à deux études de cas non triviales : l'environnement de réingénierie Moose (10 packages - 180 classes), et Morphic (61 packages - 346 classes) (voir Figure 8). Pour l'application Moose, nous avons découvert un certain nombre de cycles que les développeurs de Moose ont corrigé en conséquent.

Nous rapportons ici un exemple tiré de l'analyse de Morphic. Supprimer les cycles dans Morphic est un travail conséquent. Parfois des mécanismes d'enregistrements ou des plugins manquent et empêchent d'éliminer certaines dépendances. Morphic est clairement une application dont le packaging est à revoir car elle contient beaucoup de cycles (49 cycles directs).

Prenons le package Morphic-Kernel. Ce package, comme son nom l'indique, est le noyau de Morphic. Il ne devrait normalement avoir aucun cycle avec d'autres packages. Or, comme le montre la Figure 8, Morphic-Kernel est en cycle direct avec 17 packages (voir Figure 10).

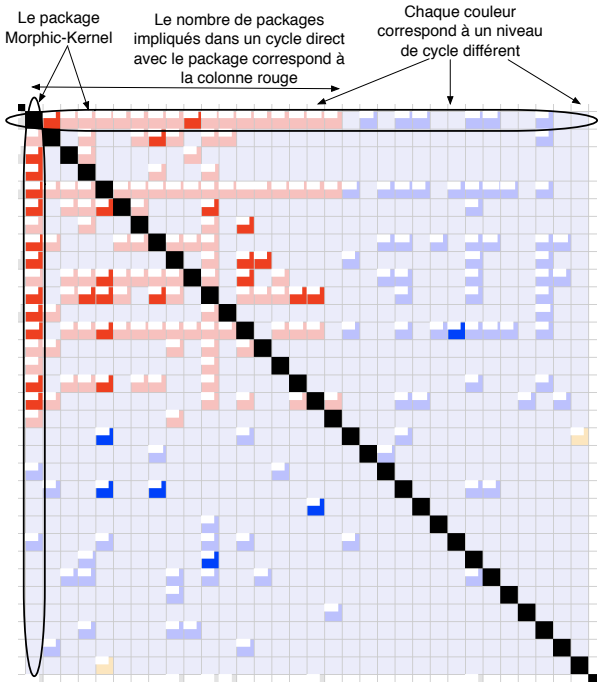


FIG. 10 – *Visualisation de différents niveaux de cycles pour le package Morphic-Kernel*

Avant d'étudier et de corriger d'autres packages, il nous est apparu intéressant d'examiner celui-ci. Par exemple, la Figure 12 montre un cycle entre Morphic-Kernel et MorphicExtras-Demo (appelé respectivement Kernel et Demo par la suite). La Figure 12 est faite de la façon suivante : on observe en fond une partie de la DSM appliquée sur Morphic. Les cases représentant les liens entre les packages Kernel et Demo sont entourées d'un cercle et agrandies. Les liens ayant pour origine Kernel sont représentés par la case rouge vif et les liens ayant pour origine Demo sont représenté par la case rouge claire. La structure du contenu des cases est expliquée dans une section précédente. Par exemple, pour la case rouge claire, il y a 72 liens dont 3 héritage (H) et 69 invocations (I), il y a 6 classes (45 méthodes) sources et 2 classes (40 méthodes) cibles. Cela représente 27% des classes (34% des méthodes) de Demo et 25% des classes (3% des méthodes) de Kernel. Les boîtes jaunes sont des popups apparaissant lorsqu'on survole la case, elles représentent la même information en version détaillée.

Ces informations nous aident à comprendre la structure de Morphic sans voir le code. En effet, sans connaître le contenu du package, on peut penser par son nom que MorphicExtras-Demo est un supplément à Morphic et qu'en plus il correspond à un package de démonstration. Pourtant, il est en cycle avec Kernel. En analysant notre DSM, on observe par les différences de couleurs rouge que Kernel fait moins accès à Demo que l'inverse. Le contenu de la case de Kernel indique un seul accès à Demo. En survolant la case, une popup s'affiche avec des informations complémentaires : la classe TheWorldMainDockingBar est la classe posant problème. Plus précisément, c'est la méthode fillDockingBar : qui fait l'appel au package Demo. Nous

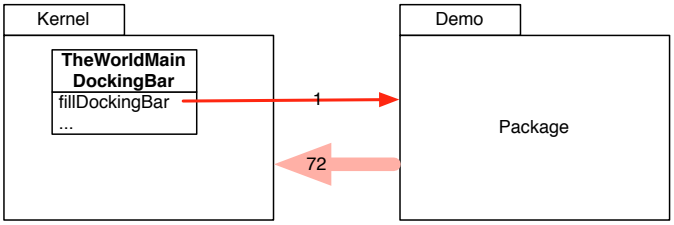


FIG. 11 – *exemple de lien avec le package Kernel*

pouvons faire la même analyse avec le package Demo, mais elle n'est pas nécessaire dans ce cas là : Kernel n'est pas censée faire un appel à ce package.

Cette analyse nous place devant trois possibilités : soit la référence est faite par du code mort et on enlève la référence, soit on déplace la classe concernée dans Demo en vérifiant que d'autres packages n'y font pas référence, soit on déplace la classe dans un package plus approprié.

L'analyse de la DSM nous montre que ce n'est pas du code mort. La classe TheWorldMainDockingBar utilise d'autres packages (par exemple MorphicExtra-Flaps), eux aussi en cycle avec Kernel. Il n'est donc pas intéressant de déplacer cette classe dans Demo, mais cette analyse nous montre qu'elle n'a pas sa place dans Kernel.

Ainsi, notre outil de DSM enrichie permet d'analyser rapidement et efficacement les cycles entre les packages sans regarder le code source et sans vraiment connaître l'application.

5 Limites de l'EDSM

Bien qu'améliorée, l'EDSM présente plusieurs limites pour lesquelles nous souhaitons trouver des solutions afin de rendre l'outil plus efficace dans la remodularisation.

- La première est la difficulté de localiser les cases symétriques. Dans la Figure 12 par exemple, il est difficile d'identifier les deux cases correspondant à un cycle lorsque ces cases sont éloignées de la diagonale. Il est envisagé dans les travaux futurs d'offrir un affichage supplémentaire des cases deux à deux.

- La deuxième est l'affichage des informations dans les cases. L'affichage textuel n'est pas un vecteur idéal pour la transmission d'information. Les valeurs 10% et 90% s'affiche de la même manière actuellement, alors que leur importances sont différentes. Il conviendrait de les différencier. Nous envisageons donc par la suite d'étudier une présentation graphique des informations textuelles.

- La troisième limite est le manque d'informations sémantiques. Cette limite n'est pas spécifique à notre approche mais commune aux approches de visulisations d'information structurelle. Pour l'instant, les informations dans les cases n'utilisent que des informations purement structurelles comme l'héritage et les références. Or dans une phase de réingénierie nous avons besoin de comprendre les fonctionalités, informations plus

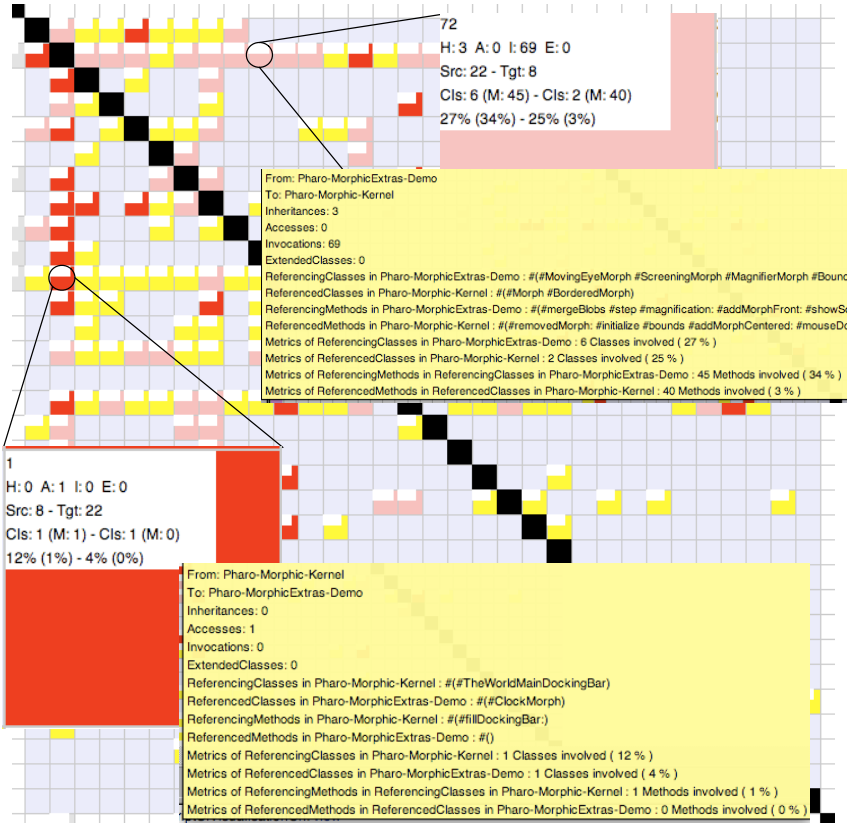


FIG. 12 – *analyse du package Kernel*

sémantiques que structurelles. Nous aimerions pouvoir annoter les dépendances avec ce type d'informations, qui pourrait être donné par le réingénieur.

6 Conclusion

Ce papier propose une amélioration des matrices de dépendance (DSM - Dependency Structure Matrix), une approche qui identifie les dépendances entre les packages logiciels. Une DSM organise les références entre les packages dans une table matricielle. Plusieurs algorithmes existent déjà pour réordonner cette matrice et ainsi révéler l'architecture logicielle.

Après avoir dressé une liste de limites des DSM traditionnelles , nous avons proposé l'amélioration de l'information contenue dans une DSM et l'amélioration de la visualisation des DSM. D'une part, le contenu des cellules est enrichi avec le type et la diffusion des références ainsi que le nombre de classes invoquées. D'autre part, plusieurs couleurs sont utilisées pour distinguer les cycles directs et indirects.

Grâce à ces améliorations, la compréhension et l'analyse d'applications logicielles deviennent plus facile et ainsi plus rapide. Des travaux sont en cours pour améliorer encore la visualisation des informations au sein d'une case.

Egalement, dans des travaux futurs, il serait très utile de pouvoir voir directement dans la matrice les conséquences des changements appliqués à l'application sans devoir modifier le code source. Ainsi, la matrice deviendrait une interface pour l'exécution de restructurations (refactorings) simples comme le déplacement d'une classe vers un autre package.

Remerciements. Le travail présenté a été développé dans le cadre du projet ANR COOK : Réarchitecturisation des applications industrielles objets (JC05 42872).

Références

Bergel, A., S. Ducasse, et O. Nierstrasz (2005). 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)*, New York, NY, USA, pp. 177–189. ACM Press.

Demeyer, S., S. Ducasse, et O. Nierstrasz (2002). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann.

Dong, X. et M. Godfrey (2007). System-level usage dependency analysis of object-oriented systems. In *ICSM 2007 : IEEE International Conference on Software Maintenance*, pp. 375–384.

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

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

Gebala, D., S. Eppinger, et M. Cambridge (1991). Methods for analyzing design procedures. *Design Theory and Methodology*.

Langelier, G., H. Sahraoui, et P. Poulin (2005). Visualization-based analysis of quality for large-scale software systems. In *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, New York, NY, USA, pp. 214–223. ACM.

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

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

Sangal, N., E. Jordan, V. Sinha, et D. Jackson (2005). Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pp. 167–176.

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

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

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

Summary

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. A number of algorithms 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 important information in their visualization to fully support a reengineering effort. For example, they do not clearly qualify or quantify problematic relationship. It is difficult to get a package centric point of view. Independent cycles are often merged. In this paper we enhanced DSM with enriched cell by showing contextual information information about (i) the kinds of references made (inheritance, class accesses..), (ii) the proportion of entities referencing, (iii) the proportion of entities referenced. We distinguish independent cycles and stress cycles using coloring information.This work has been implemented on top of the *Moose* open-source reengineering environment and the Mondrian visualization framework. It has been applied to non-trivial case studies such as the *Morphic UI* framework available in open-source Smalltalk *Squeak* and *Pharo*. Results have been implemented in the *Pharo* programming environment.