Utilisation des bases de données orientées graphe comme référentiels de modèles

Thierry Millan

Institut de Recherche en Informatique de Toulouse, Université de Toulouse
118 Route de Narbonne, F-31062 Toulouse Cedex 9, France
thierry.millan@irit.fr

RÉSUMÉ. *L'utilisation de modèles dans un contexte de production requiert de pouvoir les vérifier et les transformer afin de les exploiter au mieux, et ce, quels que soient leur taille et les outils devant les manipuler. Pour ce faire, ils doivent être sauvegardés, ainsi que leur description (les métamodèles), dans ce que l'on nomme un référentiel. Actuellement, ces référentiels stockent les modèles et leur métamodèle dans des formes propres aux différents outils qui les manipulent, ce qui rend difficile leur partage dans le cadre d'un travail collaboratif par exemple. Cet article présente une nouvelle approche pour la réalisation de référentiels de modèles basés sur une forme indépendante des plates-formes les manipulant. Cette approche utilise les bases de données graphe dont la structure est proche de celles des modèles et des métamodèles. Cet article inclut une expérimentation réalisée sur notre plate-forme NEPTUNE avec le système de gestion de bases de données graphe Neo4J.*

ABSTRACT. *The use of models in a context of production requires being able to verify them and transform them to exploit them at best, and this whatever is their size and tools that must handle them. To do it, they have tobe store das well as their description (metamodels), in what we name repositories. At present, these repositories of models store the models and their metamodels in forms specific to the various tools which handle them, what returns difficult their sharing within the scope of a collaborative work for example. This paper presents a new approach for the realization of repositories of models based on an independent type of platforms handling. This approach is based on the use of graph databases whose structure is close to those models and metamodels. This article includes an experiment carried out on our NEPTUNE platform with the management of the Neo4J graph database system.*

MOTS-CLÉS : *référentiel, modèle, métamodèle, bases de données graphes, NoSQL.*

KEYWORDS: *repository, model, metamodel, graph databases, NoSQL.*

DOI:10.3166/TSI.35.695-719

1. Introduction

L'ingénierie dirigée par les modèles (IDM) a pour objectif de rendre les modèles opérationnels à l'aide de transformations (Jézéquel *et al.,* 2012). Un modèle présente un système selon certains points de vue, à un niveau d'abstraction facilitant par exemple la conception et la validation de cet aspect particulier du système (Jézéquel *et al.,* 2012). Dans l'objectif d'exploiter ces modèles, il est nécessaire que ceux-ci soient dans un formalisme commun permettant de les décrire. Ce formalisme est lui-même un modèle que l'on nomme métamodèle. Il en résulte une relation entre le modèle et son métamodèle nommée relation de conformité. Un modèle est conforme à un métamodèle si chacun de ses éléments est instance d'un élément du métamodèle, et s'il respecte l'ensemble des propriétés (e.g contraintes d'invariant) exprimées sur le métamodèle (Jézéquel *et al.,* 2012).

Pour être exploitables, ces modèles doivent être manipulés de manière analogue au code source d'une application, c'est-à-dire qu'ils doivent pouvoir être échangés, partagés, vérifiés et transformés. Afin de faciliter leur manipulation, il faut d'une part les sérialiser, et, d'autre part, les conserver dans un référentiel permettant leur manipulation à travers différents outils. La notion de référentiel n'est pas nouvelle en IDM, car les premiers travaux sur cette thématique ont débuté lors de la réalisation des premières plates-formes de manipulation de modèles (Steinberg *et al.,* 2009) et (Thorsten, 2005). Ces travaux consistaient à définir des référentiels basés sur un langage de programmation, en l'occurrence Java, langage utilisé pour le développement des principales plates-formes de manipulation de modèles : Eclipse et Netbeans. Ces travaux consistaient à exprimer le métamodèle sous la forme d'un ensemble de classes du langage hôte et les modèles sous la forme d'instances de ces classes. Il y a donc un lien fort entre les métamodèles, les modèles et le langage hôte. Le support des référentiels basés sur netbeans/JMI n'étant plus actif, seule l'approche proposée par Eclipse/EMF (Steinberg *et al.,* 2009) subsiste, mais elle souffre d'un manque de performance pour la manipulation de modèles conséquents. Ce manquement est mentionné, entre autres, dans (Barmpis et Kolovos, 2014 ; Benelallam *et al.,* 2014 ; Pagán *et al.,* 2015 ; Stepper, 2016). Une des raisons avancées sur ces problèmes de performances réside dans l'utilisation du mécanisme d'introspection qui est un mécanisme peu performant en temps d'exécution. Il devient alors essentiel de limiter l'utilisation de l'introspection et de permettre des temps de parcours des modèles satisfaisants afin d'offrir des temps de réponse acceptables pour le concepteur lors de l'application de règles de vérification et de transformation sur des modèles conséquents. De plus, si les modèles sont conséquents, il devient aussi important d'optimiser les échanges entre le disque et la mémoire surtout si les modèles ne peuvent pas être complétement montés en mémoire vive.

Nous verrons dans cet article que le prérequis que représente EMF pose différents problèmes de performances liés à sa conception même car il utilise massivement l'introspection et qu'il monte les modèles en mémoire. Nous envisageons donc une nouvelle approche non plus basée sur l'utilisation d'EMF,

mais sur l'utilisation directe d'un système de bases de données orientées graphe. En effet, ce type de base de données supporte de fait les structures de graphes avec des algorithmes d'accès aux données et de parcours optimisés, mais offre aussi des outils de bases de données tels que les transactions, la gestion des droits d'accès et la répartition des bases sur plusieurs serveurs. Le choix de ce type de base de données est aussi dû au fait qu'un modèle ou un métamodèle peut être vu comme un graphe.

Cet article s'articulera autour de six sections. Après cette introduction, nous présenterons dans la deuxième section un rapide état de l'art concernant les référentiels de modèles. Au cours de la troisième section, nous montrerons comment il est possible d'exprimer des métamodèles et des modèles sous forme de graphes. Nous poursuivrons dans la quatrième section par la présentation de la mise en œuvre d'un référentiel pour notre plate-forme NEPTUNE avec le système de gestion de base de données Neo4J. Dans la cinquième section, nous discuterons des apports de notre approche par rapport aux approches existantes. Dans la sixième section, nous nous concentrerons sur un autre avantage de l'utilisation de bases de données comme référentiel de modèles qui est la définition d'un socle fonctionnel pour le travail collaboratif. En effet, l'utilisation de bases de données offre, de facto, la possibilité de gérer les droits d'accès sur les différents éléments manipulés et la réalisation de vues. Pour finir, nous aborderons dans la conclusion quelques perspectives pour ce travail.

2. Les référentiels de modèles

L'OMG définit un standard d'échange de modèles textuel XMI (OMG, 2014). Toutefois, cet XMI souffre du fait qu'il offre peu d'outil pour effectuer des opérations sur les métamodèles et les modèles. En particulier, il est difficile d'exprimer des contraintes telles que les WRF (well-formedness rules) qui précisent la sémantique des métamodèles et contraignent donc les modèles qui leur sont conformes. Pour ce type de vérification, l'OMG offre un langage dédié à l'expression de contraintes OCL (OMG, 2013) qui n'est pas utilisable directement sur des modèles exprimés en XMI. De plus, bien qu'il soit possible de transformer des modèles exprimés en XMI, la plupart des langages de transformation sont basés sur OCL par exemple ATL (Bruneliere, Fortin, *et al.*, 2015 ; Jouault et Kurtev, 2006), kermeta (Jézéquel *et al.*, 2009) et plus généralement ceux basés sur la norme QVT (OMG, 2011a). Il est donc nécessaire de présenter les modèles et les métamodèles exprimés en XMI dans un format permettant les traitements tels que la vérification de contraintes OCL et les transformations ce qui induit généralement le chargement complet dudit modèle dans un formalisme plus exploitable pour ce type de langage. Nous proposons dans ce papier une autre approche dans laquelle nous considérons XMI comme un vecteur d'échange de données entre outils de l'IDM et non comme une solution satisfaisante pour assurer la persistance des données de manière performante.

Des formats de représentation des modèles et des métamodèles existent, mais ils sont tous basés sur la création d'interfaces Java permettant la manipulation des modèles et des métamodèles depuis ce langage. Ce sont les formats proposés par

Sun (maintenant Oracle) pour la plate-forme Netbeans et par la fondation Eclipse pour sa plate-forme Eclipse. La solution, proposée par la fondation Eclipse, consiste à générer un schéma de classes Java et des instances de ces classes (Steinberg *et al.,* 2009) à partir de fichiers XMI. Sun a aussi défini un standard (JMI) pour manipuler des modèles et des métamodèles depuis une application Java en utilisant la représentation XMI (Oracle, 2002 ; Thorsten, 2005). Il est toutefois à noter que le format proposé par Sun n'est plus supporté. Dans ces deux approches, il y a traduction dans un langage de programmation en l'occurrence Java ce qui rend ces implantations très liées au langage sous-jacent. L'approche XMI/JMI a été utilisée, entre autre, dans le projet ModFac (Blanc *et al.,* 2005) où les auteurs privilégient l'utilisation d'un référentiel en Java par rapport à l'utilisation d'un référentiel construit sur une base de données, car cette dernière leur semble plus complexe à mettre en œuvre. On peut considérer cette remarque comme pertinente si on se focalise, comme les auteurs de l'article, sur l'utilisation de bases de données relationnelles. En effet, les bases de données relationnelles reposent sur un modèle rigide (le modèle relationnel) et sur l'opération de jointure pour la reconstruction de l'information qui est une opération extrêmement coûteuse. Toutefois, l'approche base de données offre de substantiels avantages que nous allons détailler par la suite.

3. Référentiel de modèles et bases de données

Comme nous l'avons vu précédemment, plusieurs solutions existent pour sauvegarder et manipuler des modèles et leurs métamodèles. La solution naturelle lorsque l'on veut sauvegarder des données est de recourir à l'utilisation d'un système de gestion de bases de données qui offre un ensemble de services permettant la persistance des données, le partage des données et éventuellement le contrôle d'accès à celles-ci. Elles assurent aussi l'indépendance physique c'est-à-dire l'indépendance des structures de stockage par rapport aux structures de données en tenant compte seulement des critères de performances et de flexibilités d'accès. In fine, les systèmes de bases de données permettent aux développeurs de s'abstraire de la gestion des chemins d'accès aux données en ajoutant une couche d'abstraction supplémentaire par rapport à l'utilisation de fichiers. Elles assurent aussi l'indépendance logique en permettant à chaque groupe d'utilisateurs de voir les données comme il le souhaite. Il en résulte la possibilité de faire évoluer la vue de chaque groupe en fonction de leurs besoins spécifiques. Enfin, elles sont garantes de la cohérence, du partage et de la sécurité des données. Cette sécurité vise à les protéger contre les accès non autorisés ou mal intentionnés (Gardarin, 2003, p. 23‑29). Dans le cas de ce travail, les critères nous semblant pertinents sont l'indépendance physique qui va nous permettre de profiter des optimisations d'accès aux données, l'indépendance logique pour travailler uniquement sur des parties de modèles et enfin les fonctionnalités de partage et de sécurité des données pour le travail collaboratif.

Depuis les années 1980, le modèle de bases de données prédominant est le modèle relationnel (Codd, 1970) qui est basé sur la notion de relation et d'association entre les relations. Ce modèle est, en raison de son ancienneté,

parfaitement outillé et il existe par ailleurs un grand nombre de systèmes de gestion de bases de données, qu'ils soient commerciaux ou libres. Tous ces systèmes proposent un ensemble de fonctionnalités permettant d'assurer la définition des données et leur manipulation. Toutefois, la représentation des données et les bases mêmes du modèle relationnel font qu'il est peu ou mal adapté à des données fortement connectées entre elles. En effet, dans ce modèle, les données sont éclatées en plusieurs relations pour éviter la redondance d'informations et reconstruites dynamiquement via des jointures. Cette opération, bien qu'optimisée, est fort coûteuse surtout lorsque les données sont fortement connectées et que le nombre de jointures est important pour reconstruire l'ensemble de l'information. Toutefois un système de bases de données ne se limite pas à la seule persistance des données, il permet aussi un contrôle d'accès aux données via la gestion des utilisateurs et la définition de sous-schémas au travers des vues. Un administrateur de bases de données peut alors attribuer des privilèges à une partie des données, afin d'en limiter l'accès, en fonction de l'utilisation faite de celles-ci et du droit des utilisateurs à les accéder (Gardarin, 2003).

Dans le cas qui nous intéresse, un modèle ou un métamodèle est constitué d'un ensemble d'éléments connectés entre eux par des relations que sont l'héritage, les associations et les liens (OMG, 2011b). Cette structure fait que l'utilisation de systèmes de bases de données relationnelles comme système de persistance semble peu pertinente et surtout inefficace. Cette inefficacité est due au nombre de jointures nécessaires pour reconstruire l'information et donc naviguer dans les modèles et les métamodèles (Barmpis et Kolovos, 2014). Il devient donc nécessaire d'introduire de nouveaux types de systèmes de bases de données prenant en compte les besoins de navigation dans des données fortement structurées. Ces nouveaux systèmes de bases de données ne font plus de la non-duplication de l'information un dogme, et permettent d'autres associations entre les données que les associations par valeur qui nécessitent l'opération de jointure. Ces systèmes non relationnels sont regroupés sous la dénomination de bases de données NoSQL pour Not Only SQL (Bruchez, 2013). Elles ne remettent pas en cause le langage d'interrogation SQL, mais promeuvent également d'autres langages. Ces nouveaux systèmes ont pour vocation de gérer de très gros volumes de données dans différents domaines d'activité. Ils ne sont pas nécessairement basés sur des fondements mathématiques comme les bases de données relationnelles, mais ils sont optimisés pour des besoins particuliers tels que la gestion des données issues des réseaux sociaux, la gestion des données commerciales ou la gestion de gros volumes de documents.

Dans le cas des modèles et des métamodèles, le volume de données n'est évidemment pas comparable, mais la structuration des données peut s'apparenter à la structuration des données manipulées par les réseaux sociaux. En effet, les réseaux sociaux manipulent des données structurées en graphe et nécessitent donc une optimisation des parcours dans ces graphes.

Dans notre cas, les modèles et les métamodèles, qui ne sont ni plus ni moins que des modèles pour un niveau d'abstraction supérieur (Jézéquel *et al.,* 2012) peuvent être vus comme des graphes, ayant comme nœud des objets ou des classes, et différents types d'arcs, comme ceux représentant la relation d'héritage, les

associations entre classes et les liens entre les instances de ces mêmes classes. Ces graphes peuvent être stockés et manipulés en utilisant des systèmes de bases de données NoSQL : les systèmes de bases de données graphe. Bien que ce type de système soit relativement ancien (antérieur au modèle relationnel (Gardarin, 2003), il a trouvé un regain d'intérêt, d'une part avec l'avènement des réseaux sociaux, et, d'autre part, avec l'accroissement des performances des ordinateurs.

Une base de données graphe est une base de données dont les éléments sont constitués de nœuds et d'arcs orientés entre les nœuds. Les nœuds et les arcs peuvent posséder des propriétés mettant en relation un nom (attribut) et une valeur. Certains de ces systèmes permettent de définir des labels pour les nœuds et les arcs. Ces labels servent à regrouper des éléments à la manière des types dans les langages de programmation (Van Bruggen, 2014). Il existe deux différences fondamentales avec les bases de données relationnelles qui sont que ces bases de données n'exigent pas la présence d'un schéma de base pour modéliser les données et que la valeur *null* n'existe pas ; une propriété non renseignée n'apparaît pas dans l'élément auquel elle est censée être attachée. De plus elles sont basées sur la théorie des graphes et la recherche opérationnelle. Cette base mathématique permet la mise en œuvre d'algorithmes optimisés pour le parcours de graphe et la recherche des plus courts chemins (Berge, 1983). Dans le cas des modèles, les algorithmes de recherche ne présentent pas d'intérêt, car la recherche du plus court chemin n'a pas de sens. Toutefois, il existe d'autres mécanismes intéressants comme la recherche de nœuds adjacents ou le calcul des chemins pour, par exemple, rechercher tous les ancêtres d'une classe dans l'arbre d'héritage.

(BA, 2015) présente plus d'une vingtaine de systèmes de gestions de bases de données orientées graphe. Après l'étude de plusieurs de ces systèmes, nous avons isolé deux d'entre eux qui sont OrienDB et Neo4J. Des études concernant ces deux systèmes ont déjà été menées entre autres par (Barmpis et Kolovos, 2014). Toutefois ces études prennent en compte principalement des aspects liés aux performances. Ces aspects bien qu'importants, ne sont pas ceux que nous avons voulu mettre en exergue, car notre critère principal est d'avoir un système simple pouvant s'intégrer rapidement et simplement dans notre plate-forme. En effet, l'outil choisi doit posséder une documentation riche et une communauté très réactives. À partir de ces critères, l'étude présentée dans (Vergnes, 2015) nous a incité à choisir Neo4J qui en plus des critères ci-dessus propose une version embarquée nous semblant plus simple à mettre en œuvre pour une première expérimentation.

Neo4J est un système créé en 2000 par la société Neo Technology. C'est un système de gestion de bases de données qui peut fonctionner soit en standalone, soit en cluster master/slave (Vergnes, 2015). Neo4J fournit en standard un ensemble d'algorithmes de parcours entre deux nœuds tels que les algorithmes de dijkstra et A*. Chaque nœud et chaque arc peut posséder des propriétés et des labels qui permettent d'assigner des rôles ou des types. L'utilisation de labels permet de définir plusieurs types pour une même donnée, ce qui permet de définir, pour une même entité, plusieurs points de vue (Van Bruggen, 2014). Pour manipuler les éléments qu'elle contient Neo4J propose deux approches en standard ; soit on utilise les opérations présentes dans les différentes bibliothèques fournies avec le système, soit

on utilise le langage de manipulation des données appelé cypher. Neo4J offre des bibliothèques pour différents langages, dont une pour le langage Java, langage avec lequel est développée notre plate-forme. Il est aussi à remarquer que comme les bases de données relationnelles, Neo4J supporte les propriétés ACID (Gardarin, 2003, p. 37-38) qui sont des propriétés importantes dans l'optique d'un travail collaboratif. Une présentation plus complète de Neo4J peut être trouvée sur la page web suivante (Neubauer, s. d.)

Comme nous l'avons expliqué précédemment, un système de gestion de bases de données orientées graphe manipule des nœuds et des arcs. Malheureusement, les éléments utilisés pour représenter des modèles et des métamodèles sont plus riches que ces deux seuls concepts, il faut donc enrichir la sémantique des nœuds et des arcs pour stocker les modèles et leurs métamodèles. Pour ce faire, nous allons nous appuyer sur les labels qui permettent en quelque sorte de typer les nœuds et les arcs. La différence entre un label et un type tel qu'on le conçoit dans les langages de programmation est qu'un nœud ou un arc peut avoir plusieurs types ce qui permet de filtrer les éléments du graphe en fonction du point de vue que l'on veut privilégier.

Dans la section suivante, nous allons montrer comment nous représentons des métamodèles et des modèles sous forme de nœuds et d'arcs.

4. Représentation d'un métamodèle et d'un modèle sous forme de graphe

Comme nous l'avons décrit précédemment nous allons maintenant présenter notre approche pour sauvegarder les métamodèles et les modèles dans une base de données graphe.

4.1. Transformation d'un métamodèle sous forme d'un graphe

Un métamodèle est constitué de paquetages, d'énumératifs, de métaclasses, de rôles, d'associations et de relations d'héritage**.** Nous allons, à travers deux exemples, montrer comment s'effectue la génération des nœuds et des arcs. Pour ce faire nous nous appuierons sur des exemples simples pour étayer nos propos.

La figure 1 montre un premier exemple composé d'un énumératif, de deux classes reliées par une association possédant à son extrémité deux rôles (*role1* et *role2*).

La notion de paquetage est un élément structurant qui contient soit des classifieurs, et donc des relations entre classifieurs, soit des paquetages. Comme ce concept n'est que structurant, il n'est pas nécessaire de le faire apparaître en tant que nœud ou arc. Nous avons donc choisi d'utiliser la hiérarchie de paquetages uniquement en tant qu'espace de nom. Dans la figure 1, le nom de la classe *Classe1* sera préfixé par *paquetage1.paquetage2*.

Dans les métamodèles se côtoient deux types de classifieurs : les métaclasses et les énumératifs. Dans la base de données, les classifieurs sont des nœuds du graphe

possédant des labels. Pour ce faire, nous introduisons quatre labels : *CLASS* pour identifier une métaclasse, *ENUM* pour identifier un énumératif, *ENUM_CLASS* pour indiquer que le type n'est pas un type primitif et enfin un label représentant le nom de l'élément en tenant compte de son espace de nom.

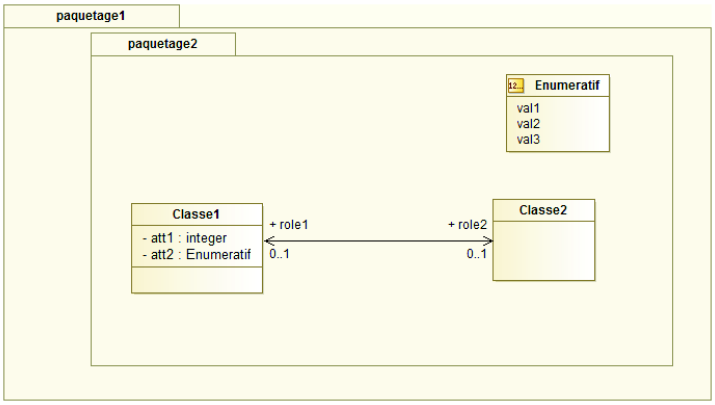

Figure 1. Métamodèle 1

Il en résulte donc qu'une classe possède au moins trois labels : son nom, *CLASS* et *ENUM_CLASS*. Cette multiplication de labels est nécessaire pour mettre en œuvre les outils d'optimisation de parcours de graphe que le système de gestion de bases de données propose. Les attributs de la classe forment les propriétés du nœud. À la différence des propriétés d'un nœud d'un graphe, les propriétés d'une classe sont typées généralement par des types atomiques ou des types énumératifs. A contrario, les propriétés d'un graphe sont faiblement typées, elles ne supportent que les types atomiques voire des tableaux de type atomiques. Il en résulte l'impossibilité d'avoir des propriétés dont les types sont des énumératifs, car ce type de données n'est pas supporté par Neo4J. Pour mettre en œuvre les types énumératifs, nous avons envisagé trois solutions. La première est d'inclure le nom du type dans le nom de l'attribut (par exemple *Visibilitykind:visibility* pour indiquer que l'attribut *visibility* est de type *VisibilityKind*). La deuxième est de stocker la valeur et le type dans un tableau de chaînes de caractères à deux dimensions. L'élément d'indice 0 est le type, et celui de rang 1 est la valeur. La troisième est d'utiliser un arc partant du nœud représentant la métaclasse vers le nœud représentant l'énumératif. Pour indiquer le nom de l'attribut on ajoute deux propriétés sur l'arc, une indiquant le nom de l'attribut de la métaclasse dont l'énumératif est le type et l'autre la valeur énumérée de l'attribut. Dans notre proposition, nous avons décidé de faire apparaître le type en tant que préfixe du nom de l'attribut.

Pour un énumératif, on utilise les propriétés du nœud pour définir les éléments le constituant. Chaque élément est une propriété et la valeur associée, un numéro d'ordre commençant à zéro.

Si on prend l'exemple de la figure 1, la classe *Classe1* possède les labels suivants : *CLASS*, *ENUM_CLASS*, *paquetage1.paquetage2.Classe1*. *Classe1* possède deux attributs qui seront matérialisés par deux propriétés qui s'appelleront respectivement *integer:att1* et *paquetage1.paquetage2.Enumeratif:att2*. En ce qui concerne l'énumératif il aura comme labels : *ENUM*, *ENUM_CLASS*, *paquetage1.paquetage2.Enumeratif.* Le nœud représentant l'énumératif possédera trois propriétés *val1*, *val2*, *val3* ayant respectivement les valeurs *0*, *1* et *2*.

Les classes sont liées soit par des associations qui relient les rôles des classes, soit par des liens d'héritage. Dans notre cas, seules les associations binaires, c'est-à-dire reliant uniquement deux classes entre elles, sont pertinentes, elles peuvent donc être représentées par des arcs. Les rôles portant des informations sont représentés par des nœuds reliés aux classes correspondantes par des arcs ayant comme label *ASSOCIATIONEND*. Chaque rôle va posséder un label *ASSOCIATIONEND* et contenir les attributs suivants *name, lower, upper, ordered* et *kind* représentant respectivement le nom du rôle, la cardinalité minimum, la cardinalité maximum, si le rôle est ordonné ou non et son type (faux s'il n'y a pas de type et vrai s'il est de type *composition*). Si l'on reprend l'exemple de la figure 1, il existe une association entre *classe1* et *classe2*. Cette association sera matérialisée par deux nœuds représentant les deux rôles (*rôle1* et *rôle2*) et trois arcs ; un reliant *classe1* à *rôle1* labellisé *ASSOCIATIONEND*, un reliant *rôle1* à *rôle2* labellisé *RELATIONSHIP* et un reliant *classe2* à *rôle2* labellisé *ASSOCIATIONEND.* Le nœud représentant *rôle1* possède les propriétés *name, lower, upper, ordered* et *kind* avec pour chacune les valeurs respectives *rôle1, 0, 1, none* et *none*. Le nœud *rôle2,* lui, possède pour les mêmes attributs les valeurs suivantes : *rôle2, 0, 1, none* et *none*.

Il existe un type particulier d'associations qui est la classe-association. Une classe-association est simultanément une classe et une association comme le montre la figure 2 avec la classe-association *E*. Dans ce cas, on transforme la classe-association en une double association entre *C* et *E* puis entre *E* et *D.*

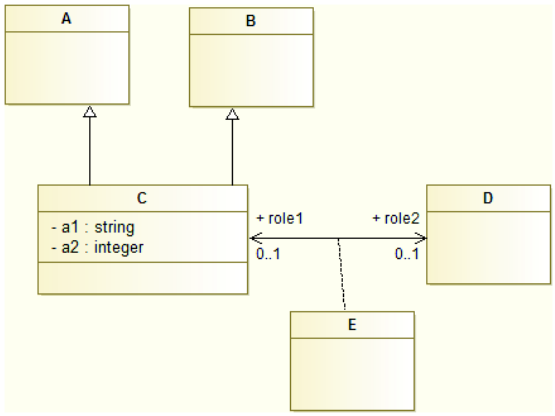

Figure 2. Métamodèle 2

La figure 3 présente le résultat de la transformation de la classe-association de la figure 2 en une association entre trois classes. Une fois cette transformation effectuée, on applique les règles de passage que nous avons présentées précédemment pour les associations. La seule différence est que l'on positionne pour les deux arcs implantant les deux associations entre *rôle1* et *rôle2* la propriété *estClassAssociation* à vrai. Cette propriété indique que le nœud compris entre deux relations possédant cet attribut à vrai représente une classe-association.

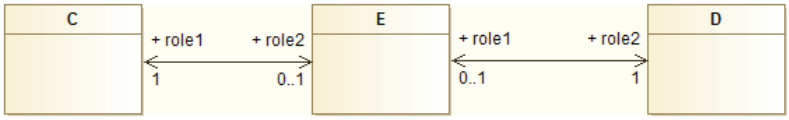

Figure 3. Transformation d'une classe-association

Les liens d'héritage, eux, sont représentés par des arcs reliant deux nœuds (du nœud représentant la classe fille vers le nœud représentant la classe mère). Ces arcs possèdent le label *INHERIT*. Si on reprend l'exemple de la figure 2, il y aura création d'un arc de *C* vers *A* et de *C* vers *B*. Ces deux arcs seront labellisés *INHERIT*.

Afin d'illustrer le mécanisme de transformation, nous proposons dans la figure 4 et dans la figure 5 le résultat de la transformation du métamodèle 1 et du métamodèle 2.

Dans la figure 4, le nœud violet représente l'énumératif (en haut sur la figure), les nœuds roses représentent les classes (à gauche et à droite de la figure) et les nœuds de couleurs beiges (au centre de la figure) les *rôles*. Afin d'éviter tout risque de conflit de nom, les labels sont constitués de la manière suivante : « __ »TYPE_LABEL « _ *17061967*_ », par exemple *__ASSOCIATIONEND_17061967_* pour les arcs de type *ASSOCIATIONEND*.

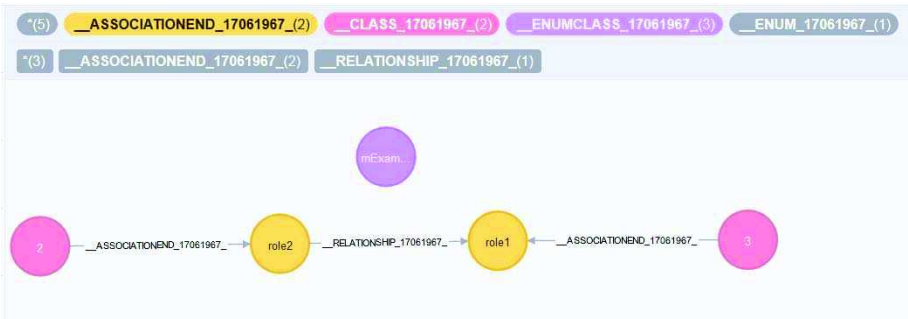

Figure 4. Représentation du métamodèle 1 sous forme de graphe dans Neo4J

Dans la figure 5, les nœuds roses sont reliés entre eux par des arcs de type *INHERIT* (nœud n° 10 qui représente la classe *C* héritant des classes *A* et *B* – nœuds n° 8 et n° 9), les nœuds beiges par des arcs de type *RELATIONSHIP* (nœud *role1* et *role2*) et enfin les nœuds roses et beiges par des arcs de type *ASSOCIATIONEND* (nœud n° 11 et nœud *role 1*). Il est aussi à remarquer que le sens des arcs de type *RELATIONSHIP* et *ASSOCIATIONEND* ne présente aucun intérêt, car nous naviguons généralement à travers les arcs dans les deux sens. Enfin, le nœud n° 12 représente la classe-association *E* et sa mise en œuvre telle qu'elle est présentée figure 3.

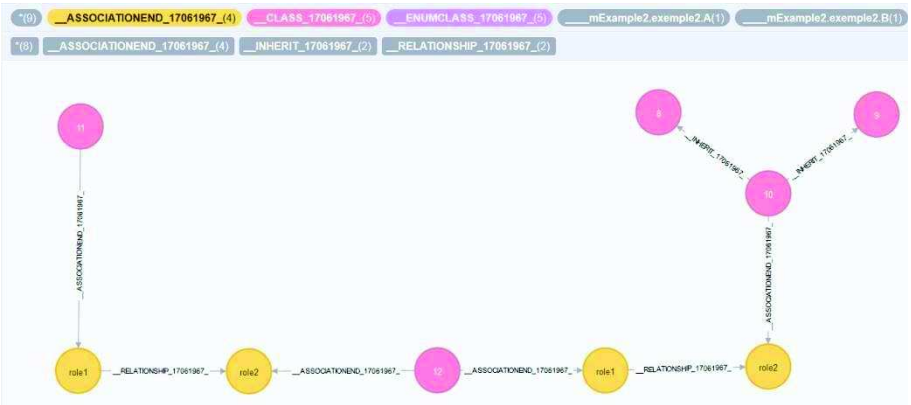

Figure 5. Représentation du métamodèle 2 sous forme de graphe dans Neo4J

La figure 6 montre la valeur des propriétés pour le nœud *role2* relié au nœud n° 12. On remarque que ce rôle se trouve sur une classe-association, propriété *estClasseAssociation* à vrai, qu'il n'est ni ordonné, ni une composition. De plus *lower* et *upper* contiennent la multiplicité minimum et maximum du rôle.

Une fois le métamodèle chargé, il faut pouvoir aussi charger des modèles conformes à ce métamodèle. C'est ce que nous allons présenter dans la prochaine section.

ordering	unordered
kind	false
upper	1
lower	0
estClasseAssociation	true

Figure 6. Propriétés du nœud role 2 relié au nœud n° 12

4.2. Transformation d'un modèle sous forme d'un graphe

Toute instance est représentée par un nœud relié par des arcs labellisés *INSTANCEOF* aux nœuds représentant sa métaclasse. La principale difficulté dans les modèles réside dans la gestion de la hiérarchie de classes liée à sa métaclasse. De manière ascendante, il faut rechercher tous les attributs ayant des valeurs par défaut et les ajouter au nœud s'ils n'y sont pas déjà. Dans l'exemple de la figure 7, considérons l'attribut *atta* de la classe *A* qui a comme valeur par défaut *titi*, il faut donc, lors de la création du nœud correspondant à l'objet *c2,* récupérer la valeur par défaut de l'attribut *atta* de la classe *A.*

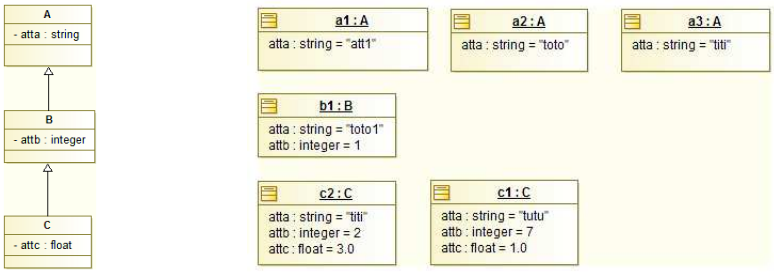

Figure 7. Exemple d'un diagramme de classes et d'un diagramme d'objets correspondant

De manière descendante, il faut récupérer tous les nœuds instance des classes filles pour récupérer toutes les instances d'une classe donnée. Toujours dans l'exemple de la figure 7, si l'on veut accéder aux instances de la classe *A*, il faut récupérer les instances *a1*, *a2*, *a3*, *b1*, *c1* et *c2*, et donc, accéder aux nœuds en lien *INSTANCEOF* avec les classes *B* et *C*. Pour les attributs, toutes les propriétés de la métaclasse et de ses ancêtres au sein du nœud représentant l'instance sont recopiées.

La figure 8 présente le résultat de la transformation du diagramme de classes et des diagrammes d'objets de la figure 7 sous forme de graphe. Les nœuds en gris (en bas de la figure) correspondent aux différentes instances du modèle objet. Ils sont reliés à leurs métaclasses respectives via des arcs de type *INSTANCEOF.* Les nœuds roses représentant les métaclasses (nœuds n° 17, 18 et 19) sont reliés entre-eux par des arcs de type *INHERIT* représentant les liens d'héritage présents dans le diagramme de classes correspondant.

Les exemples proposés précédemment nous proposons figure 9 un exemple plus complet issus du métamodèle 1.5 (OMG, 2003) et qui illustre les liens entre les instances des métaclasses.

La figure 9 présente deux éléments d'un modèle et leurs métaclasses associées. Les deux instances sont reliées par un arc labellisé *LINK.* En ce qui concerne leurs métaclasses, elles sont liées à des rôles par des arcs labellisés *ASSOCIATIONEND.*

Ces deux rôles sont eux-mêmes reliés entre eux par un arc labellisé *RELATIONSHIP*. La recherche des valeurs du rôle *parent* revient donc, pour une instance donnée (celle de gauche en l'occurrence), à parcourir les liens labélisés :

– *INSTANCEOF* pour accéder au nœud représentant la classe dont l'instance est issue ;

– *INHERIT* et *ASSOCIATIONEND* pour rechercher le rôle *parent* dans la hiérarchie de classes dont l'instance est issue,

– *RELATIONSHIP* qui part ou arrive sur ce rôle pour en extraire l'identifiant de la relation en lien avec le rôle *parent*.

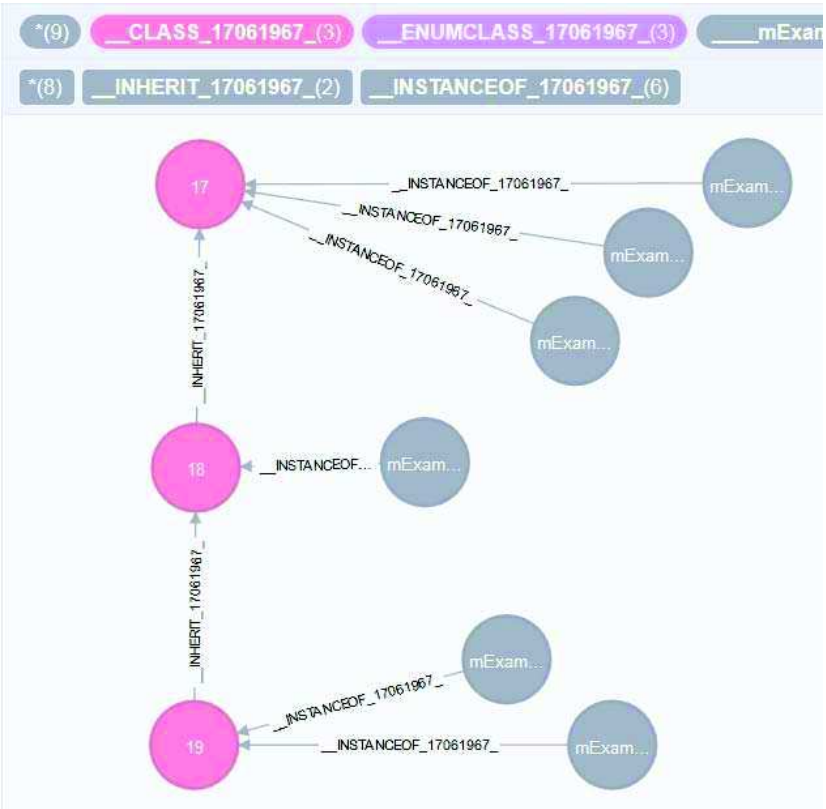

Figure 8. Exemple d'un diagramme de classes et des diagrammes d'objets correspondant mis sous forme de graphe

Une fois cet identifiant isolé, on recherche tous les arcs labellisés *LINK-identifiant*. Ce parcours de graphe est effectué directement par le système de gestion de bases de données à travers ses bibliothèques, et ne nécessite donc que peu de

programmation et surtout aucune mise en œuvre d'algorithmes de parcours de graphe.

Par exemple dans la figure 9, l'arc sélectionné (❶) a comme identifiant *306* (❷), le lien reliant les deux objets se nomme donc *_LINK_17061967_306* (❸).

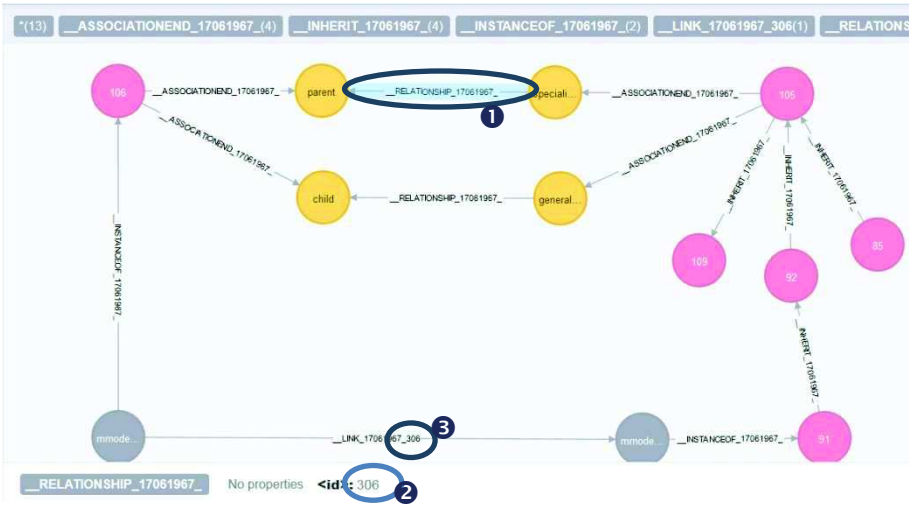

Figure 9. Exemple d'un graphe contenant des liens entre les nœuds représentant des instances

Code 1 présente les différents appels aux méthodes de la bibliothèque Neo4J permettant à partir d'une instance de rechercher les instances liées au rôle *a*. La ligne 2 contient la définition des différents liens qui devront être traversés. Nous pouvons voir que les liens *INSTANCEOF*, *INHERIT* et *ASSOCIATIONEND* seront parcourus dans le sens de la source vers la cible. *BreadthFirst* indique que nous privilégions un parcours en largeur d'abord. La ligne 3 affecte au parcours défini ligne 2 un évaluateur qui permet de personnaliser ce parcours en prenant en compte les contraintes de l'utilisateur dans notre cas cet évaluateur *monEvaluator* stoppe le parcours dès que le nœud représentant le rôle *nomRole* est trouvé.

Code 1. Recherche des valeurs d'un rôle

```
1 :     TraversalDescription accesAuxRoles
2 :     accesAuxRoles =  new GraphDatabaseFactory().newEmbeddedDatabase(
          new java.io.File(DB_PATH)).traversalDescription().breadthFirst().
            relationships(INSTANCEOF, Direction.OUTGOING ).
            relationships(INHERIT, Direction.OUTGOING).
            relationships(REL_ASSOCIATIONEND,  Direction.OUTGOING)  ;
3 :     ResourceIterator<Node> role = accesAuxRoles.evaluator(new
          monEvaluator(nomRole)).traverse( instanceNode ).nodes().iterator()  ;
```

Les figures 8 et 9 montrent que dans notre approche métamodèles et modèles cohabitent au sein d'un même environnement. La figure 10 synthétise notre approche. Le niveau métamodèle et le niveau modèle se retrouvent donc dans un même espace qui est un graphe. On ne manipule plus que deux concepts de base : des nœuds et des arcs.

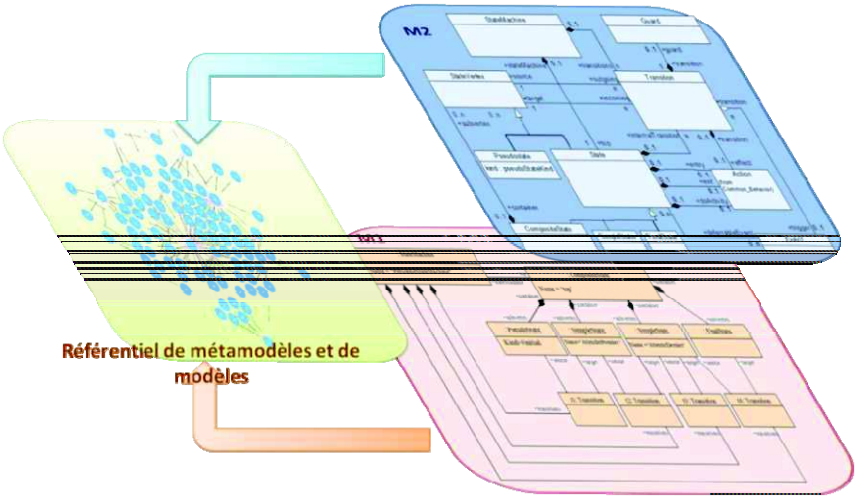

Figure 10. Synthèse du passage métamodèles/modèles vers un graphe

Le principal avantage de l'utilisation de deux concepts pour tout représenter est que la navigation dans les modèles et les métamodèles se fait de manière uniforme par le parcours de nœuds d'arcs en arcs.

Après avoir présenté le chargement des métamodèles et des modèles sous forme de graphe, nous allons montrer la mise en œuvre que nous avons réalisée sur notre plate-forme NEPTUNE. Nous ne présenterons pas ici de code, nous nous limiterons à montrer les adaptations que nous avons apportées pour intégrer ce nouveau référentiel.

5. Mise en œuvre de Neo4J dans la plate-forme NEPTUNE

La première version de la plate-forme NEPTUNE fut réalisée pour le projet IST nommé NEPTUNE (IST-1999-2001). Les objectifs de cette plate-forme étaient le développement de différents outils pour vérifier des modèles et pour générer la documentation associée. À l'issue de l'évaluation de celle-ci, diverses améliorations nous ont semblées pertinentes. Une seconde version de la plate-forme (Millan, 2013) a donc été développée entièrement par l'équipe MACAO de l'institut de recherche en informatique de Toulouse (IRIT). Le but de cette nouvelle version, que nous nommerons NEPTUNE II, est de fournir un évaluateur de règles OCL (Millan *et al.,*

2009 ; OMG, 2013) qui est le langage d'expression de contraintes de modèles, quels que soient le modèle et le métamodèle considérés. Il permet d'exprimer des invariants de classes, des pré et post conditions de méthodes. Ces contraintes peuvent s'exprimer aussi bien sur les modèles que sur des métamodèles, mais la plate-forme NEPTUNE II évalue uniquement les règles du niveau métamodèle.

L'architecture de cette plate-forme est basée sur un ensemble de six composants communiquant via des interfaces comme le montre la figure 11.

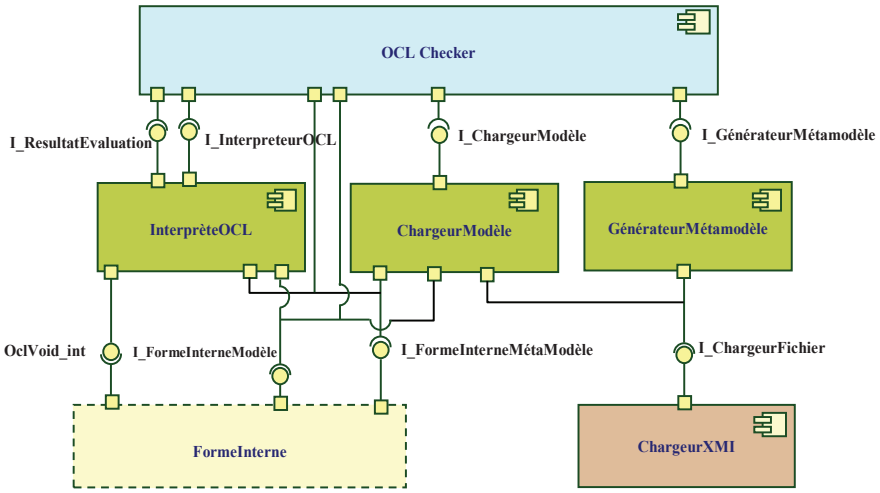

Figure 11. Architecture de la plate-forme NEPTUNE II

Lors de la première version de la plate-forme NEPTUNE II, nous avions opté pour un référentiel constitué d'un ensemble de classes et d'interfaces Java représentant les différents éléments du métamodèle. Les modèles eux étaient implantés en tant qu'instance des classes du métamodèle. Ces métamodèles sont rassemblés au sein d'un composant nommé *FormeInterne.* L'interrogation des métamodèles et des modèles s'effectue au travers des deux interfaces du composant *FormeInterne.* Ces deux interfaces permettent l'accès aux éléments des métamodèles pour la première et aux éléments des modèles pour la seconde. Nous avions donc opté dans cette version pour une approche similaire à celle de Netbeans et d'Eclipse. Les premières expérimentations montrent que cette première version souffre d'une incapacité à supporter de gros modèles. De plus, cette version a été développée afin de travailler sur des modèles et des métamodèles centralisés au sein d'une même machine et donc par un seul et unique concepteur à la fois.

L'objectif de ce deuxième développement est de remplacer la forme interne constituée de classes, d'interfaces et d'instances par des nœuds et des arcs. Dans cette nouvelle version, les fichiers objets java sont remplacés par une base de données orientée graphe ; en l'occurrence Neo4J (Neo4J Team, 2015). Cette

migration doit entraîner un ensemble minimal de modifications de la plate-forme et en particulier des interfaces de consultation des métamodèles et des modèles. Cette nouvelle implantation offre un double avantage : permettre la manipulation de gros modèles et offrir un socle technologique pour partager les métamodèles et les modèles. En effet, le propre des bases de données est d'offrir un socle technique pour le partage de données entre plusieurs utilisateurs. De plus, les bases de données NoSQL permettent la mise en œuvre de solutions réparties sur différentes machines.

Lors de cette expérimentation, nous avons privilégié la version embarquée de Neo4J pour simplifier l'installation de la base de données et avoir une version de démonstration qui peut être rapidement déployée. Dans cette nouvelle approche, seul un composant *GénérateurMétamodèle* a nécessité une refonte complète, car c'est celui qui permet de charger les métamodèles dans le référentiel. Pour ce composant, il a fallu remplacer les opérations de génération de classes et d'interfaces par des opérations de création de nœuds et d'arcs. Grâce à l'utilisation des interfaces entre les composants, il n'y a pas eu d'effet de bord. Les autres modifications apportées à l'implantation de la plate-forme concernent la mise en œuvre des deux interfaces de *FormeInterne* et à celle d'*Ocl_Void_int*. *Ocl_Void_int* est l'interface qui représente le type OCL *Void*. Pour les classes implantant ces interfaces, nous avons dû remplacer le corps des méthodes faisant appel à l'introspection pour interroger les métamodèles par des appels aux opérations du système Neo4J.

Une fois ces modifications réalisées, nous avons validé notre solution en comparant les résultats obtenus entre la version utilisant un référentiel à base de classes et d'interfaces Java et celle utilisant un référentiel Neo4J. Cette validation consiste à vérifier que la nouvelle mise en œuvre offre au moins un niveau de performance équivalent à la première mise en œuvre. En effet, l'utilisation d'un système de bases de données, quel qu'il soit, nécessite la traversée de couche logicielle impactant plus ou moins les performances. Pour effectuer les prises de temps, la plate-forme possède une fonctionnalité que nous avons utilisée. Cette fonctionnalité est aussi bien utilisée pour la génération des métamodèles, que pour le chargement des modèles, que lors de l'évaluation des règles OCL.

Tableau 1. Comparaison entre deux versions de NEPTUNE II (en seconde)

	Classes-Interfaces	Neo4J
Création d'un métamodèle	10,506	5,586
Chargement d'un modèle (taille du fichier XMI 40 Mo)	60,174	41,558
Évaluation d'une requête complexe	1,535	2,527
	0,790	0,921
	0,771	0,788

Le tableau 1 présente les différentes prises de temps que nous avons réalisées avec deux versions de la plate-forme NEPTUNE, une utilisant un référentiel de

modèles à base de classes et d'interfaces, et l'autre un référentiel Neo4J. Les tests utilisés ont été développés à l'origine dans le cadre de la thèse de Cédric Bouhours (Bouhours, 2010). Ils s'appliquent à des modèles de taille significative et visent à valider des requêtes OCL générées automatiquement. Ces requêtes permettent de détecter des ensembles d'éléments de modélisation pouvant être remplacés par des patrons de conception tels que ceux édictés par E. Gamma (Gamma *et al.,* 1994). Ce protocole d'évaluation n'a d'autre intérêt que d'utiliser un modèle de taille respectable et surtout une requête OCL complexe.

Nous avons aussi réalisé une nouvelle série de tests en utilisant cette fois l'étude de cas proposée lors du cinquième séminaire international concernant les outils basés sur les graphes en 2009 (GraBaTs Van Gorp *et al.,* 2009). Cette étude propose un métamodèle et cinq modèles (set0, set1, set2, set3 et set4) de tailles différentes allant d'un modèle dont le fichier XMI fait 8Mo à un modèle dont le fichier XMI fait 661Mo. Les premiers tests que nous avons effectués mettent en exergue le nombre de nœuds et d'arcs générés pour les cinq modèles proposés.

La figure 12 montre qu'avec Neo4J le temps de chargement est linéaire et ne dépend que de la taille du fichier contenant le modèle. La version utilisant les classes et les interfaces ne fonctionne, elle, que pour les deux premiers modèles (set0 et set1). Le temps de chargement semble croître beaucoup plus vite qu'avec l'implantation utilisant Neo4J. Nous avons réussi à charger le modèle set2, mais avec un temps de 60005,013 secondes (soit presque 17 heures). Ce temps est considérable, mais s'explique par le fait que la création d'objets ainsi que l'appel de méthodes par ce mécanisme ne profitent pas des optimisations que réalise le compilateur Java lors de la compilation d'un programme. Cette impossibilité d'optimiser la création et les appels de méthodes est sûrement aussi la cause de l'échec du chargement des modèles set2, set3 et set4 dans les expérimentations relatées dans (Barmpis et Kolovos, 2014). Dans la figure suivante, nous nous intéressons aux données créées dans la base de données.

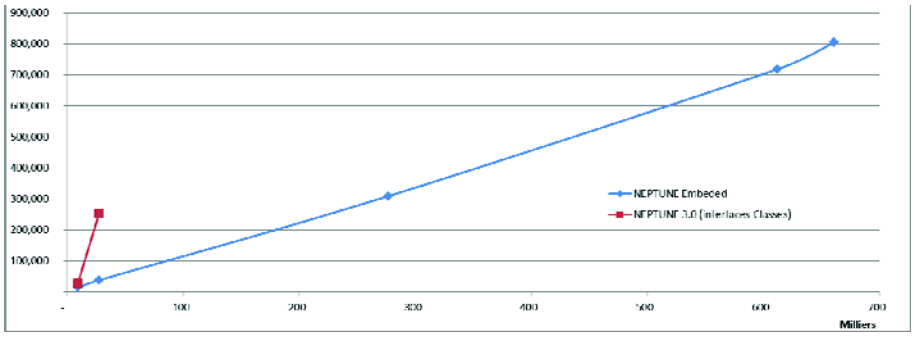

Figure 12. Temps de chargement comparé entre la version utilisant Neo4J et celle utilisant des classes et des interfaces Java

La figure 13 montre que le nombre d'arcs est plus important que le nombre de nœuds ce qui semble logique si l'on considère que chaque nœud peut être l'origine de plusieurs arcs (un nœud représentant un élément de modèle, il est relié à une métaclasse par un lien de type *INSTANCEOF* et éventuellement à une ou plusieurs autres instances via des liens de type *LINK*. On peut remarquer aussi un accroissement linéaire du nombre des arcs et des nœuds lorsque la taille du fichier contenant le modèle croit. Cet accroissement est lui aussi logique si on considère que l'accroissement de la taille des fichiers est proportionnel au nombre d'éléments de modélisation qu'il contient et que le nombre d'éléments influe sur le nombre de nœud et d'arc. Le nombre de nœuds est exprimé en millions de nœuds et la taille des fichiers XMI en kilo-octets.

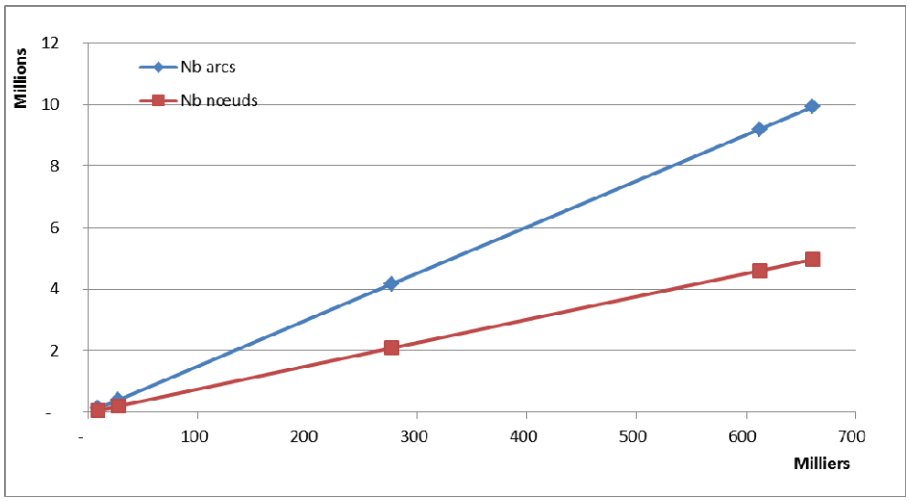

Figure 13. Nombre d'arcs et de nœuds créés en fonction de la taille du fichier contenant le modèle

Pour conclure avec ces tests, nous n'avons pas fait apparaître la taille du métamodèle celui-ci ne fait que 56 ko et que le nombre de nœuds et d'arcs qui le compose est respectivement de 480 nœuds et 790 arcs.

Ces tests montrent un fort gain lors des phases de génération des métamodèles et de chargements de modèles. Le gain lors de la génération du métamodèle est logique si l'on considère d'une part que le chargement du métamodèle dans la seconde version ne nécessite aucune compilation et que d'autre part la création des nœuds est moins complexe que la création des classes et des interfaces Java. Le gain lors du chargement des modèles est moins marqué, il est vraisemblablement le fait que, dans la première version, le chargement consiste à créer des instances de classes via le mécanisme d'introspection de Java qui est relativement coûteux en termes de performance, coût que nous avions compensé par un système de caches. Les performances lors de l'exécution de requêtes OCL complexes sont un peu moins

bonnes par rapport aux performances obtenues en utilisant le référentiel à base de classes et d'interfaces. Cette légère perte de performances est sûrement due au fait que Neo4J utilise une architecture en couches. De plus, et comme dit précédemment, Neo4J possède des mécanismes complexes pour la navigation dans des graphes qui ne sont pas pleinement exploités dans notre cas. Nous sommes conscients que ces mécanismes devront être étudiés et utilisés lors de tests plus poussés.

Dans ce qui précède, nous avons montré qu'il est possible de réaliser un référentiel de métamodèles en utilisant comme base de persistance un système de gestion de bases de données. Nous allons dans la section suivante discuter de notre proposition et en particulier faire une évaluation critique de celle-ci.

6. Discussion

Dans cette étude, deux aspects doivent être pris en considération : l'aspect graphe comme concept fédérateur des modèles et des métamodèles, et l'aspect base de données qui permet d'accéder à un certain nombre de fonctionnalités permettant de partager des modèles et des métamodèles.

Le concept de graphe est fortement basé sur un modèle mathématique ce qui confère aux graphes une algorithmique et des propriétés intéressantes entre autres pour les transformer et pour valider mathématiquement ces transformations. Ces fondements mathématiques ont servi de cadre pour formaliser les travaux sur la transformation de modèles (Tran et Percebois, 2012). L'utilisation d'un référentiel basé sur les graphes offre donc un formalisme propice à l'exploitation des travaux sur les transformations de graphes appliquées à la transformation de modèles.

L'utilisation de bases de données n'est pas nouvelle pour sauvegarder des modèles. Deux approches intéressantes pour notre étude ont fait l'objet d'expérimentations : une utilisant CDO (Stepper, 2016) et l'autre NEO4EMF (Benelallam *et al.,* 2014). CDO est une extension d'EMF permettant l'utilisation de référentiels distants pour accéder aux modèles et les manipuler. CDO supporte de multiples systèmes de persistance. À ce titre, CDO peut interagir de manière transparente avec tout type de bases de données qu'elles soient objets, relationnelles ou NoSQL. Toutefois, bien que CDO soit une solution mature, elle n'est pas adaptée lorsque le volume de données devient important (Benelallam *et al.,* 2014, p. 4 ; Pagán *et al.,* 2015). NEO4EMF est comme son nom l'indique basé sur l'utilisation de Neo4J comme base de persistance. Neo4J est transparent aux utilisateurs, car l'accès aux modèles se fait au travers d'EMF qui charge les données de Neo4J et les sauvegarde dans la base Neo4J. Il en résulte qu'EMF doit assurer la cohérence de son environnement avec les données stockées dans la base de données (Benelallam *et al.,* 2014). La solution que nous proposons n'utilise pas un modèle intermédiaire qui fait le lien entre notre API et le système de base de données. Nous avons conscience que ce choix nous prive d'un haut niveau d'abstraction, mais nous considérons que le coût pour ce haut niveau est trop préjudiciable à des performances acceptables. De plus, notre proposition évite d'avoir à assurer la cohérence entre une forme intermédiaire et la base de données. La délégation

complète au système de base de données est possible par le fait que dans notre proposition les modèles et les métamodèles sont représentés de manière identique dans la base de données et par le fait que nous avons implanté un typeur indépendant de celui du langage Java pour assurer le typage des règles OCL (Millan *et al.,* 2009). Nous n'avons donc pas besoin de charger les données en instanciant des classes Java pour manipuler les instances de modèles. Qui plus est, cette solution assure un typage correct des entités manipulées sans avoir recours au système de type Java. Ce choix est dû à notre volonté d'avoir les meilleures performances possibles lors de l'évaluation de requêtes OCL sur de gros modèles en déléguant le plus possible les traitements et la navigation dans les modèles au système de base de données qui possède des algorithmes optimisés pour ce type de traitement.

Un autre avantage d'utiliser un système de bases de données graphe est que ce type de système de gestion de bases de données n'oblige pas la définition de métaschémas. C'est une différence fondamentale avec le modèle relationnel ou le langage Java. En effet, la mise en œuvre de classes et d'interfaces Java est un travail long et fastidieux comme le montre le temps de traitement nécessaire pour générer un métamodèle ou pour charger un modèle. Le temps de génération d'un métamodèle est significatif, car cette génération nécessite la compilation de classes pour définir la structure, et cette opération requiert un temps non négligeable. Dans notre cas, l'utilisation d'un système de bases de données graphe permet, par sa souplesse, d'implanter tous les concepts des modèles et des métamodèles tels que l'héritage multiple ou la multi-instanciation sans avoir à générer au préalable une structure complexe. Cette souplesse est le fait du faible nombre de concepts manipulés par des bases de données graphe : des nœuds et des arcs. Il faut toutefois remarquer que ces systèmes sont optimisés pour des calculs de chemin dans un graphe et cette optimisation n'est pas pertinente dans notre cas, car tous les arcs ont le même poids, une sémantique différente et les chemins à traverser sont relativement courts.

Si, par certains aspects, le fait que le nombre de concepts soit faible présente des avantages, il s'avère que le faible typage des concepts implique la mise en place d'un gestionnaire de type complet pour pallier l'absence de celui du système sous-jacent. En effet, Netbeans et Eclipse s'appuient sur le typage de Java pour le typage des métamodèles. Lors de l'utilisation d'une base de données graphe, il est impossible de s'appuyer sur celui fourni par le système de base de données, car celui-ci est quasiment inexistant hormis si l'on considère les labels.

Nous venons de voir dans cette section les possibilités qu'offrent les bases de données pour sauvegarder de gros modèles. La réalisation de tels modèles ne peut être le fait d'une seule équipe et va donc nécessiter la collaboration de différentes équipes potentiellement distribuées. C'est en partant de ce constat que nous allons dans la section suivante présenter quelques pistes pour mettre en place le travail collaboratif en exploitant au mieux les fonctionnalités offertes par les systèmes de bases de données. Nous visons dans cette partie à proposer ces nouvelles fonctionnalités qui ne sont que peu considérées par les plates-formes actuelles qui offrent un nombre limité de fonctionnalités pour le travail collaboratif.

7. Vers plus de collaboration pour réaliser les modèles

Par essence même, les systèmes de bases de données offrent un ensemble de fonctionnalités permettant le partage et la gestion des accès aux différentes données présentes en leur sein. Ces capacités de partage présentent un intérêt si l'on considère de très gros modèles potentiellement réalisés par des équipes réparties géographiquement et dans des sociétés différentes. Les bases de données NoSQL, à la différence des systèmes de bases de données relationnelles, sont pensées pour être massivement réparties. Par exemple, on peut considérer que chaque société possède son référentiel et met à disposition d'autres participants à un projet les éléments de modélisation pertinents à celui-ci, et uniquement eux, au travers de vues et d'une gestion fine des droits d'accès sur ceux-ci. Pour ce faire, on peut associer la gestion des droits d'accès et des langages du type QVT (OMG, 2011a) permettant de définir des vues mettant en exergue des parties de modèles en fonction de l'utilisation qui en sera faite. C'est ce mécanisme qui est un des fondements pour la maîtrise des accès à l'information dans les bases de données.

Neo4J présente un mécanisme de gestion de transactions permettant la modification en toute sécurité des données. Une transaction est une suite d'opérations permettant de faire passer la base de données d'un état stable à un autre état stable. Une transaction est généralement réversible, c'est-à-dire qu'il est possible de revenir au dernier état stable si nécessaire. Ce mécanisme est essentiel, en particulier lors de la modification de modèles, afin d'annuler une transformation dont le résultat ne serait pas celui escompté. Eclipse a mis en œuvre un tel mécanisme (Philippot, 2014) pour les mises à jour des modèles au sein de sa plate-forme Eclipse. Toutefois, cette gestion de transaction n'est pas distribuée et ne s'applique pas à la modification de modèle effectuée par plusieurs développeurs simultanément dans des lieux géographiquement répartis. Une expérimentation doit donc être conduite en déployant Neo4J sur différentes machines et en utilisant les extensions que nous avons incluses dans notre interprète OCL (Millan, 2013). Cette extension permet entre autres de réaliser des vues sur des modèles.

La dernière fonctionnalité nécessaire pour permettre le travail collaboratif est la gestion des utilisateurs et des droits d'accès sur les données. Dans notre cas, il faut constater que Neo4J n'offre pas en natif une vraie gestion des utilisateurs et des droits d'accès. Ce support est prévu à terme, mais pour le moment il faut installer une extension pour gérer les utilisateurs, mais force est de constater que Neo4J n'offre pas actuellement en natif une vraie gestion des utilisateurs et de leurs droits d'accès, cela implique qu'il faut mettre en œuvre son propre gestionnaire d'utilisateur pour étendre les fonctionnalités offertes par l'extension. Cette mise en œuvre au coup par coup nuit grandement à la portabilité du système proposé.

8. Conclusion et perspectives

Nous avons présenté dans cet article une première étude concernant la mise en œuvre d'un référentiel de modèles à partir d'une base de données graphe. Après avoir présenté les solutions actuellement usitées dans les plates-formes gérant les

modèles, nous avons proposé une solution pour sauvegarder des modèles et leurs métamodèles dans des bases de données graphe. Cette solution présente différents avantages : en particulier un métamodèle étant en réalité un modèle d'un niveau d'abstraction supérieur, nous ne proposons plus qu'une seule représentation pour les modèles quel que soit leur niveau d'abstraction. De plus, cette solution offre la persistance pour les gros modèles et un socle fonctionnel pour le travail collaboratif. Toutefois, l'utilisation d'un système de base de données a des répercussions sur les temps de traitement et en particulier sur les temps liés à la vérification des modèles. En effet, nous n'avons pas, lors de nos expérimentations, réussi à atteindre un temps de réponse proche de celui obtenu lors de l'exécution de la même requête sur notre plate-forme utilisant un référentiel à base de classes et d'interfaces Java. Bien que la raison semble inhérente au fait que l'utilisation d'un système de base de données engendre la traversée de multiples couches logicielles pour atteindre la donnée, cet état de fait n'est pas acceptable, et des investigations en particulier sur la communication entre la plate-forme et la base de données doivent être menées. L'utilisation d'un référentiel basé sur les graphes permet aussi de mettre en place de manière efficace la notion de vue sur les modèles. En effet, l'utilisation d'une représentation à base de graphe permet de dissocier le modèle de son métamodèle et donc d'accéder aux modèles indépendamment du métamodèle ce qui permet à un utilisateur l'accès à l'un et non forcément à l'autre. Cette dissociation du modèle et de son métamodèle est plus difficile à réaliser lorsque le modèle est une instance de classes Java.

À partir de ces premiers travaux, nous envisageons quatre pistes d'études qui nous semblent pertinentes. La première est de comparer les résultats obtenus lors de la mise en œuvre de l'étude de cas proposée lors du cinquième séminaire international concernant les outils basés sur les graphes en 2009 (GraBaTs Van Gorp *et al.,* 2009) avec ceux obtenus par la mise en œuvre de ces mêmes tests sur une plate-forme Eclipse/NEO4EMF. Les expérimentations menées sur la base de cette étude de cas fournissent des résultats quantitatifs qui seront une base intéressante pour évaluer notre proposition. La deuxième est de fournir une formalisation des transformations d'un métamodèle et d'un modèle sous forme de graphe et vice-versa. La troisième piste de réflexion concerne l'expérimentation de la notion de vue. Cette notion est déjà proposée dans la plate-forme Eclipse (Bruneliere, Perez, *et al.,* 2015), mais nous désirons évaluer qualitativement l'apport de l'approche et des outils propres aux bases de données pour définir des vues sur les modèles et les métamodèles. Cette évaluation nécessite la mise en œuvre d'un contrôle d'accès aux éléments des modèles et des métamodèles afin de promouvoir efficacement le travail collaboratif. Cette évaluation nécessite d'étudier la gestion de la répartition des bases de données NoSQL. Cette répartition permet, dans le cadre du travail collaboratif, de sauvegarder des modèles au plus près des équipes qui les réalisent. Nous pourrons ainsi conjuguer la répartition, le contrôle d'accès et la notion de vue afin de permettre une telle collaboration, tout en préservant la confidentialité de tout ou partie des modèles et de leurs métamodèles. La dernière piste est de réfléchir sur la définition d'un cinquième type de bases de données qu'on pourrait nommer bases de données orientées IDM ou modèle qui reposerait pour la partie interrogation sur un langage supportant le standard QVT (OMG, 2011a) ou notre extension du

langage OCL (Millan *et al.,* 2009). Il faudrait compléter ce langage pour inclure le langage de mise à jour des données.

Bibliographie

BA. (2015, avril 20). 20+ Free and Open Source Graph Database. *Butler Analytics*.

Barmpis K., Kolovos D. (2014). Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models. *Journal of Object Technology* vol. 13, n°3, p. 1-26.

Benelallam A., Gómez A., Sunyé G., Tisi M., Launay D. (2014). Neo4EMF, A Scalable Persistence Layer for EMF Models. In J. Cabot et J. Rubin (Eds.), *Modelling Foundations and Applications*, Vol. 8569, p. 230‑241. Cham: Springer International Publishing.

Berge C. (1983). *Graphes*, Gauthier-Villars.

Blanc X., Gervais M.-P., Sriplakich P. (2005). Model Bus: Towards the Interoperability of Modelling Tools. *Model Driven Archit.* Berlin, Springer-Verlag, p. 17‑32.

Bouhours C. (2010). *Détection, explications et restructuration de défauts de conception : les patrons abîmés*. Thèse en Génie logiciel, Université de Toulouse - Toulouse III.

Bruchez R. (2013). *Les bases de données NoSQL : Comprendre et mettre en oeuvre*, Paris, Eyrolles.

Bruneliere H., R. Fortin, T. Fortin. (2015). ATL/User Guide - Introduction. *Eclipsepedia*. https://wiki.eclipse.org/ATL/User_Guide_-_Introduction

Bruneliere H., Perez J. G., Wimmer M., Cabot J. (2015). EMF Views: A View Mechanism for Integrating Heterogeneous Models. *34th International Conference on Conceptual Modeling (ER 2015)*, Stockolm, Sweden. <hal-01159205 >

Codd E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, vol. 13, n° 6, p. 377‑387.

Gamma E., Helm R., Johnson R., Vlissides J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional.

Gardarin G. (2003). *Bases de données*, Paris, Eyrolles.

Jézéquel J.-M., Barais O., Fleurey F. (2009). Model driven language engineering with kermeta. *International Summer School on Generative and Transformational Techniques in Software Engineering*, Berlin, Springer-Verlag, p. 201–221.

Jézéquel J.-M., Combemale B., Vojtisek D. (2012). *Ingénierie Dirigée par les Modèles : des concepts à la pratique,* Paris, Ellipses

Jouault, F., Kurtev I.. (2006). On the architectural alignment of ATL and QVT. In C. of ACM (Ed.), *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06)*, p. 1188‑1195.

Millan, T. (2013). *The Neptune platform*, http://neptune.irit.fr/images/download/Neptune2013/UserManualV2.pdf

Millan T., Sabatier L., Thi T.-T. L., Bazex P., Percebois C. (2009). An OCL extension for checking and transforming UML Models. *In proceedings of the International Conference*

on Software Engineering, Parallel and Distributed Systems (SEPADS'09). WSEAS Press,. p. 144‑150.

Neo4J Team. (2015). *The Neo4J Manual*, Neo4J,. http://neo4j.com/docs/stable/

Neubauer, P. (s. d.). *Graph Databases*, NOSQL and Neo4j.

OMG. (2003). *OMG Unified Modeling Language (OMG UML), UML 1.5. OMG Available Specification Version 1.5 - formal/03-03-01*.

OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. janvier 2011b. http://www.omg.org/spec/QVT/1.1/

OMG. OMG Meta Object Facility (MOF) Core Specification - OMG Available Specification Version 2.4.1 - formal/2011-08-07. 2011a. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF

OMG. Object Constraint Language, OMG Available Specification, Version 2.4 beta with CB - ptc/2013-08-13. 2013. http://www.omg.org/spec/OCL/2.4/

OMG. (2014) XML Metadata Interchange (XMI) Specification.

Oracle. JSR-000040 Java(TM) Metadata Interface API Specification 1.0 Final Release. juin 2002. < http://download.oracle.com/otn-pub/jcp/7868-jmi-1.0-fr-spec-oth-JSpec/jmi-1_0-fr-spec.pdf?AuthParam=1447708941_4b763a881f3e5d07c26fd03c10b0d4c9 >

Pagán J. E., Cuadrado J. S., Molina J. G. (2015). A repository for scalable model management. *Software & Systems Modeling* vol. 14, n° 1, p. 219‑239.

Philippot S. (2014, Avril). *La programmation transactionnelle au sein d'Eclipse*. ENG221 Information et communication pour l'ingénieur, Toulouse.

Steinberg D., Budinsky F., Paternostro M., Merks E. (2008) *EMF: Eclipse Modeling Framework* (2nd Edition) (Eclipse). Addison-Wesley Longman, Amsterdam.

Stepper, E. (2016). CDO Model Repository - Documentation - *CDO Model Repository Overview*, https://eclipse.org/cdo/documentation

Thorsten P. (2005) *Java Metadata Interface (JMI)*. http://www2.cs.uni-paderborn.de/cs/kindler/Forschung/ComponentTools/ct-seminar/Java%20Metadata%20Interface%20-%20Ausarbeitung.pdf

Tran H. N., Percebois C. (2012). Towards a Rule-level Verification Framework for Property-Preserving Graph Transformations. *Verification and validation Of model Transformations (VOLT)*, Montreal (Canada), IEEE Computer Society, p. 946‑953.

Van Bruggen, R. (2014). *Learning Neo4j*. Birmingham: Packt.

Van Gorp P., Rensink A., Levendovszky T. (2009). *GraBaTs 2009: Graph-Based Tool Contest*, http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/?page=Home

Vergnes N. (2015). *Bases de données graphes : comparaison de NEO4J et OrientDB*. ENG221 Information et communication pour l'ingénieur, Toulouse.

Article reçu le 14/04/2016

Accepté le 23/01/2017