# IFT4055 - Projet Honor

MQL à Cypher – Un Framework pour Traduire un Langage de Requêtes Spécifique au Domaine pour Interroger un Projet Versionnable en des Expressions Cypher

Philippe Gabriel - 20120600

18 décembre 2022

# Problème

Les systèmes de versions de contrôle (*Version Control System* – VCS) présentent une grande importance dans les besoins de vouloir gérer le versionnement de collections de données. Dans le domaine de l'informatique et de l'ingénierie logicielle, des systèmes comme Git représentent une très grande importance dans la pratique menant à l'intérêt industrielle du développement de grands logiciels. L'ingénierie dirigée par les modèles (*Model-Driven Engineering* – MDE) est une approche présentant plusieurs avantages dans la conception de logiciels. Cette approche permet au concepteur d'abstraire des détails des langages de programmation générals, de sorte à concentrer ses efforts dans la résolution du problème. L'usage de modèles ou de langages spécifiques au domaine (*Domain-Specific Languages* – DSL) aide l'expert à s'exprimer dans une terminologie propre à son domaine d'étude et lui permettrait de contribuer à la conception d'un logiciel sans avoir de connaissances de programmation. Un intérêt industrielle commence à se manifester pour la MDE poussant à l'adopter pour concevoir de larges projets. Un VCS orienté-ligne comme Git utilisé pour gérer le versionnement des changements apporter à un projet MDE n'est pas la meilleure approche à adopter gérer de gros projets. L'interprétation des changements sur un modèle conçu par l'expert du domaine sera au niveau de la sérialisation du modèle. Cette interprétation n'est donc pas spécifique au domaine du modèle. La sémantique riche du domaine est complètement négligé. Un VCS spécifique au domaine (*Domain-Specific VCS* – DSVCS) serait plus approprié pour des projets MDE. DSMCompare [1] permet de comprendre les différences entre modèles en terme de la sémantique d'un langage de modélisation. Il s'agit d'une approche considérant la syntaxe abstraite et concrète d'une DSL pour exprimer les différences entre modèles. Ce niveau de détection de changements est une composante cruciale dans un DSVCS. Un DSVCS devrait adhérer à différents critères tel que la définition d'une unité versionnable pour permettre la comparaison sémantique, les commandes offertes pour manipuler le projet, la possibilité d'interroger le VCS selon le projet, la méthode de stockage, etc... [2]. Pour ce projet, je présente une approche MDE basé sur la traduction d'expressions MQL – un langage de requêtes spécifique au domain (*Domain-Specific Query Language* – DSQL) pour interroger un projet versionnable – en des expressions Cypher – un langage de requêtes supporté par des bases de données de graphes Neo4j. Ce projet vient essentiellement proposer une solution quant à la composante d'interroger le VCS.

# Méthodes utilisées

La solution proposée est basée sur la MDE. Sa presque totalité a donc été rédigée dans le Eclipse Modeling Framework (EMF). Je présente, dans les sous-sections ci-dessus, les différentes étapes de la solution.

## Projet Versionnable

Afin d'interroger le VCS sur un projet versionnable, il est nécessaire de déterminer à priori les unités versionnables d'un projet. Cela est décrit dans le métamodèle d'un projet versionnable présenté à la Figure 1. Un projet versionnable a une structure assez similaire à celle des projets utilisant un VCS linéaire. Un projet consiste en des branches. Les branches contiennent des commits définissant un point où l'on désire enregistrer des changements. Les modèles constituent ici l'unité dont l'expert du domaine modifiera dans le projet MDE. Chaque modèle possède une durée de vie exprimée sous forme d'une série de snapshots.

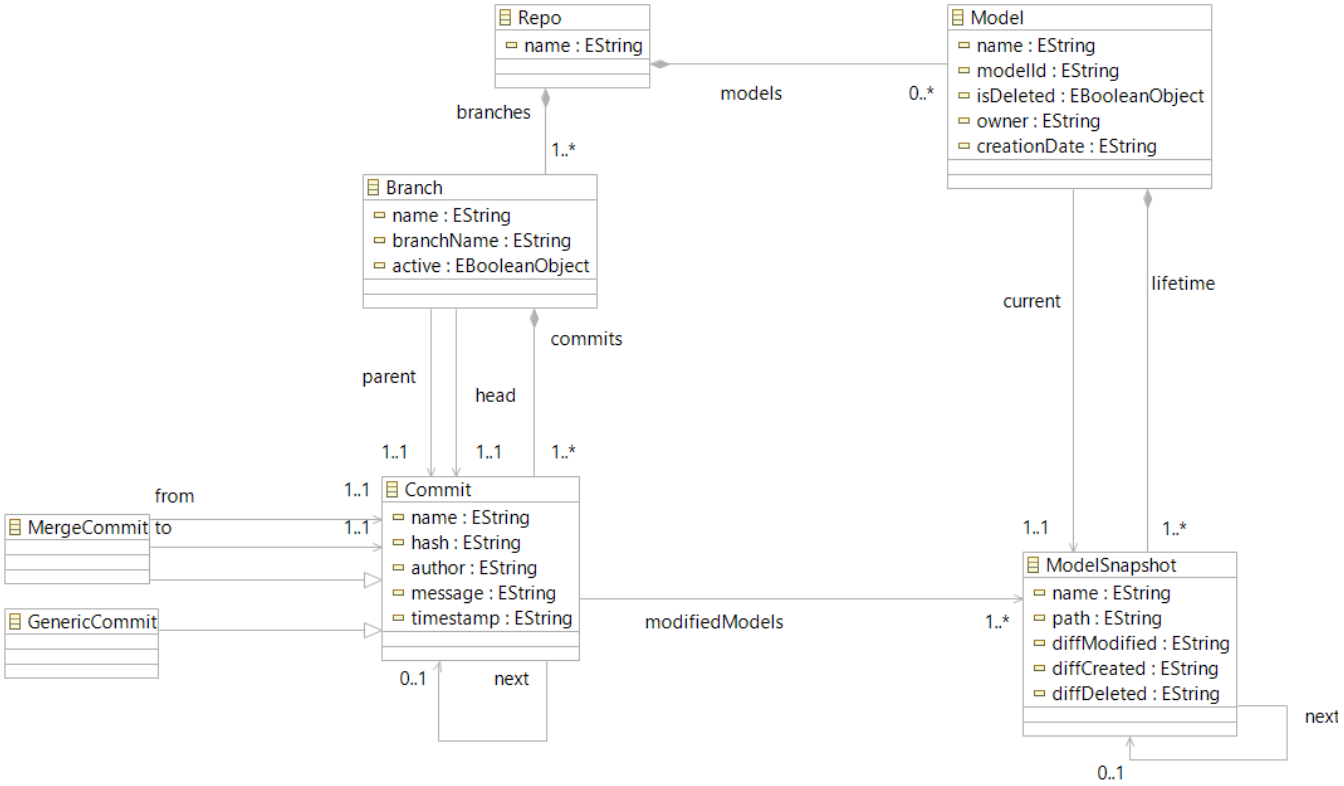


Figure 1: Métamodèle d'un Projet Versionnable

## Model Query Language - MQL

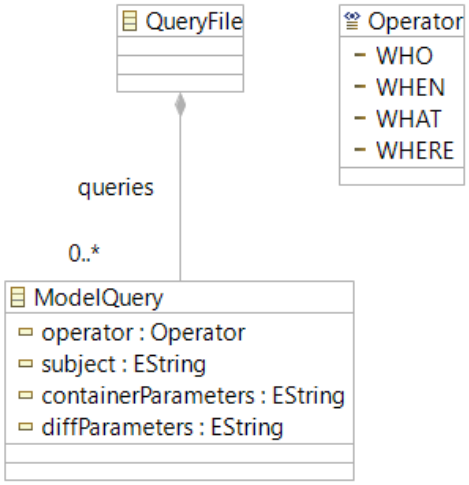


Figure 2: Métamodèle de MQL

La syntaxe abstraite du langage MQL est exprimé dans le métamodèle de MQL à la Figure 2. Une expression MQL est constituée de plusieurs requêtes. Chaque requête est composée d'un opérateur, du sujet de requête, et de paramètres pour mieux filtrer les résultats attendues. La syntaxe concrète de ce langage a été définie à l'aide de Xtext. Le Listing 1 démontre un exemple

d'expression MQL. Dans cette expression, on remarque que chaque requête est séparée par une virgule et que la dernière requête se termine par un point d'interrogation. Chaque requête commence par un opérateur. La seconde requête utilise l'opérateur `DESCRIPTION` qui s'agit ici tout simplement de sucre syntaxique pour l'opérateur `WHERE` qui a été jugé ne pas être très intuitif dans ce contexte.

```
WHO head {
  branchName = "main",
  active = "true"
},
DESCRIPTION parent {
  branchName = "b1"
},
WHEN created {
  timestamp < "2022-09-04"
} [
  "MyDomainSpecificObject.x = 19"
]?
```

Listing 1: Expression MQL

Un parseur Xtend a été implémenté pour produire un modèle MQL à partir d'une expression MQL. Ce dernier génère un modèle MQL conforme au métamodèle MQL lorsque l'expert du domaine sauvegarde le fichier où il rédigeait ses requêtes MQL.

## Transformations Modèle-à-texte

À partir d'un modèle d'un projet versionnable conforme à son métamodèle, il est ensuite possible de l'utiliser comme entrée à une transformation modèle-à-texte qui produit une requête Cypher correspondante au modèle donné. Dans le cas d'un projet versionnable, une requête pour créer une base de données de graphes est générée à partir d'un modèle de projet versionnable.

À partir d'un modèle d'une expression MQL conforme à son métamodèle, une autre transformation modèle-à-texte se sert de ce modèle comme entrée pour générer la requête Cypher correspondante pour interroger la base de données existante du projet versionnable. En prenant comme exemple la requête décrite au Listing 1, après que le parseur Xtend aie généré le modèle MQL conforme au métamodèle MQL, lorsque ce modèle est pris comme entrée à la transformation modèle-à-texte, on obtient la requête Cypher décrite au Listing 2.

```
MATCH (b1:Branch)-[h1:head]->(c1:Commit)
WHERE b1.branchName = "main" AND b1.active = "true"
RETURN c1.author;
MATCH (b2:Branch)-[p1:head]->(c2:Commit)
WHERE b2.branchName = "b1"
RETURN c2.message;
MATCH (c3:Commit)-[mm1:modifiedModels]->(ms1:ModelSnapshot)
WHERE c3.timestamp < "2022-09-04" AND "MyDomainSpecificObject.x = 19" IN ms1.
 diffCreated
RETURN c3.timestamp;
```

Listing 2: Cypher générée

## Automatisation

Afin d'automatiser le processus de création de base de données à partir d'un projet versionnable et de traduction de la requête MQL pour ensuite afficher les résultats de la requête, un Ant Workflow

a été implémenté décrivant les étapes du framework. La Figure 3 décrit les différentes étapes de celui-ci.

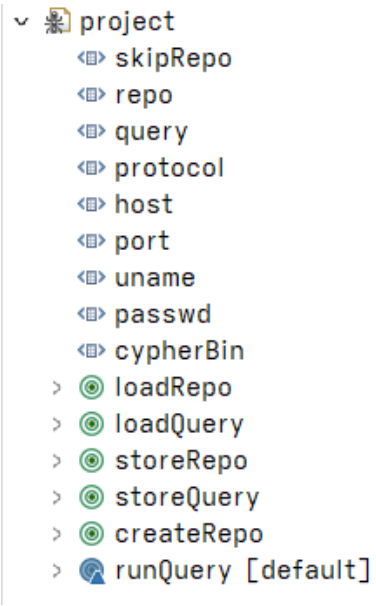


Figure 3: Outline du Workflow

Il est possible de configurer certaines propriétés avant de démarrer le workflow. La propriété `skipRepo` agit comme booléen décrivant si l'on désire ne pas créer la base de donnée (dans le cas où elle a déjà été créée). Les propriétés `repo` et `query` réfère aux noms de fichiers des modèles du projet versionnable et de la requête qui seront utilisées comme entrée aux transformations modèle-à-texte. Les propriétés `protocol`, `host`, `port`, `uname`, et `passwd` représentent les valeurs pour se connecter à la base de donnée de graphes où l'on veut entreprendre nos requêtes. La propriété `cypherBin` représente le chemin de l'exécutable `Cypher` qui permettra de rouler des requêtes dans la base de données. Le workflow est constitué de tâches à effectuer. Les tâches `loadRepo` et `loadQuery` ne font que simplement charger le modèle du projet versionnable et du modèle de la requête MQL respectivement. Les tâches `storeRepo` et `storeQuery` appliquent la transformation modèle-à-texte sur les modèles chargés. La tâche `createRepo` initialise la base de donnée à partir de la requête Cypher généré du modèle de projet versionnable. La tâche `runQuery` exécute la requête Cypher générée à partir du modèle MQL et affiche les résultats.

# Résultats obtenus

Pour décrire les résultats obtenus, je vais présenter un exemple d'utilisation. Considérant en premier lieu, le métamodèle d'une salle à manger. La Figure 4 illustre le métamodèle en question. Il s'agit d'un simple métamodèle décrivant la relation entre des tables et des chaises dans une salle. Nous supposerons que l'expert du domaine conçoit un projet de salle à manger.

```
1  WHAT head {
2    branchName = "main"
3  }?
```

Listing 3: Exemple d'expression MQL

La figure 5 décrit l'état du projet versionnable de notre expert du domaine sous forme de modèle conformant au métamodèle de projet versionnable. Celui-ci s'agit d'un projet assez simple contenant 3 modèles de salle à manger, 2 branches et 6 commits au total. On a par la suite comme exemple de requête celle présentée au Listing 3. On peut maintenant exécuter le workflow qui commence par génèrer les requêtes Cypher appropriées, initalise la base de donnée, et exécute la requête Cypher obtenue de la transformation modèle-à-texte du modèle MQL pour obtenir les résultats de la Figure 6.

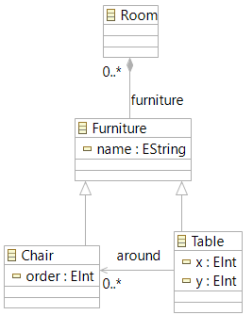


Figure 4: Métamodèle d'une salle à manger

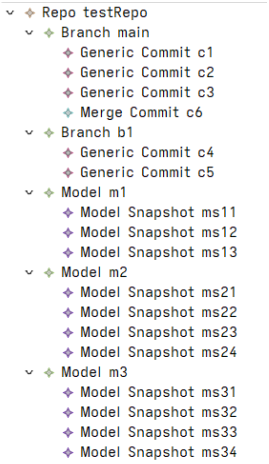


Figure 5: Modèle de Projet Versionnable

```
[exec] c1
[exec] (:MergeCommit:Commit {name: "c6", message: "Merge 6", author: "Alice", hash: "h67890", timestamp: "2022-04-01"})
```

Figure 6: Résultat de la première requête

Considérant maintenant la requête au Listing 4. Puisque nous avions déjà généré la base de données de graphes de notre projet versionnable, on s'assure de mettre la propriété `skipRepo` à `true` avant de démarrer le workflow.

```
1  WHERE model {
2    owner = "Philippe"
3  },
4  WHEN changed [
5    "MyTable.x = 3"
6  ]?
```

Listing 4: MQL Expression specific to Dining Room domain

En exécutant le workflow après avoir bien définit les valeurs des propriétés, on obtient le résultat illustré à la figure 7.

# Améliorations futures

En conclusion, j'ai présenté un langage de requête spécifique au domaine de projets versionnables pour interroger un projet versionnable sous un domaine spécifique. J'ai également présenté un framework décrivant comment les différentes composantes du projet interagissent entre elles. Il a été nécessaire de définir les unités versionnables d'un projet à l'aide du métamodèle d'un projet

```
[exec] ms1.path
[exec] "Room1.dr"
[exec] "SpecialRooms/Room2.dr"
[exec] "SpecialRooms/Room3.dr"
[exec] c1.timestamp
[exec] "2021-01-01"
```

Figure 7: Résultat de la seconde requête

versionnable et de définir les transformations modèle-à-texte nécessaire pour générer les requêtes Cypher pour interagir avec la base de données Neo4j.

En tant que travaux futurs, je prévoie ajouter un meilleur support pour la fonctionnalité d'autocomplétion offerte par Xtext au moyen d'être capable d'accéder au contenu d'un modèle existant. Un tel ajout ferait en sorte que les paramètres `diff` au sein d'une expression MQL ne sont plus fournis en tant chaînes de caractères littérales mais des références aux éléments existants du modèle spécifique au domaine dans le projet. Il permettrait également la validation de requêtes impliquant des paramètres `diff`.

De plus, je prévoie améliorer l'expressivité des requêtes MQL. À l'heure actuelle, des requêtes de base peuvent être effectuées en utilisant les différents opérateurs fournis. Il serait intéressant de permettre des requêtes plus complexes, basées sur une séquence de commits par exemple, ou fournissant des analyses statistiques.

Présentement, l'outil de workflow n'est disponible que dans Eclipse. Le possibilité d'invoquer l'outil en tant qu'utilitaire de ligne de commande, ou un fichier jar, ou même un un plugin Eclipse serait préféré pour donner l'opportunité qu'il soit intégré aux flux de travail existants. Pour ce faire, il faudrait regrouper les composantes et y inclure les dépendances appropriées pour l'exécution indépendante de EMF.

La mise en oeuvre d'un parseur de projet versionnable est une idée intéressante dans le contexte de l'inclusion d'une telle une étape dans le framework. Considérant un projet versionnable existant spécifique au domaine, qu'il soit géré par des VCS orientés ligne tels que git ou non, il serait souhaitable de générer le modèle de projet versionnable correspondant conforme au métamodèle de projet versionnable.

Le Neo Modeling Framework (NMF) [3] est un ensemble d'outils open-source conçu pour manipuler des ensembles de données ultra-volumineux dans la base de données Neo4j. Il comprend un chargeur qui a la capacité de charger un modèle Ecore et de produire une base de données de graphes. Il fournit également un générateur qui peut générer une API spécifique au domaine pour modifier un modèle spécifique. Intégrer le workflow lors de l'utilisation du MQL avec NMF pourrait fournir divers atouts. L'intégration avec NMF et la comparaison avec le flux de travail actuel est une idée future qui pourrait aider à mieux orienter les idées de ce projet.

L'intégration avec DSMCompare [1] est un autre atout souhaitable à considérer. Par exemple, il pourrait être possible d'interroger les différences sémantiques entre les modèles d'une expression MQL et récupérer le `diff_model` à partir de DSMCompare sur lequel plus de requêtes peuvent être exprimées.

# Références

[1] Manouchehr Zadahmad Jafarlou, Eugene Syriani, Omar Alam, Esther Guerra, and Juan de Lara. Dsmcompare: domain-specific model differencing for graphical domain-specific languages. *Software and Systems Modeling*, 21(5):2067–2096, October 2022.

[2] Eugene Syriani and Manuel Wimmer. A roadmap towards domain-specific version control systems. *White Paper*, September 2018.

[3] Nikitchyn Vitalii. A framework for domain-specific modeling on graph databases. *Universite de Montreal*, December 2021.