Stage encadré par Clara BERTOLISSI – 10/06/2023 au 26/07/2024
Rapport de stage condensé - DI SALVO-CILIA Pauline

Vérifier les principes du RGPD grâce aux graphes de provenance

Ce stage est la suite d'un travail initié en TER en groupe de 4 étudiant·es (CHEBRINE Ghiles, PAPA Hichame, SABOUR Zakaria, DI SALVO-CILIA Pauline).

1. Affichage de graphes de provenance

Travail sur la visualisation des graphes de provenance (affichage du graphe complet et de sous-graphes correspondant aux différents problèmes détectés).

a. Structure et démonstration

Considérations générales

La structure actuelle du prototype est la suivante :

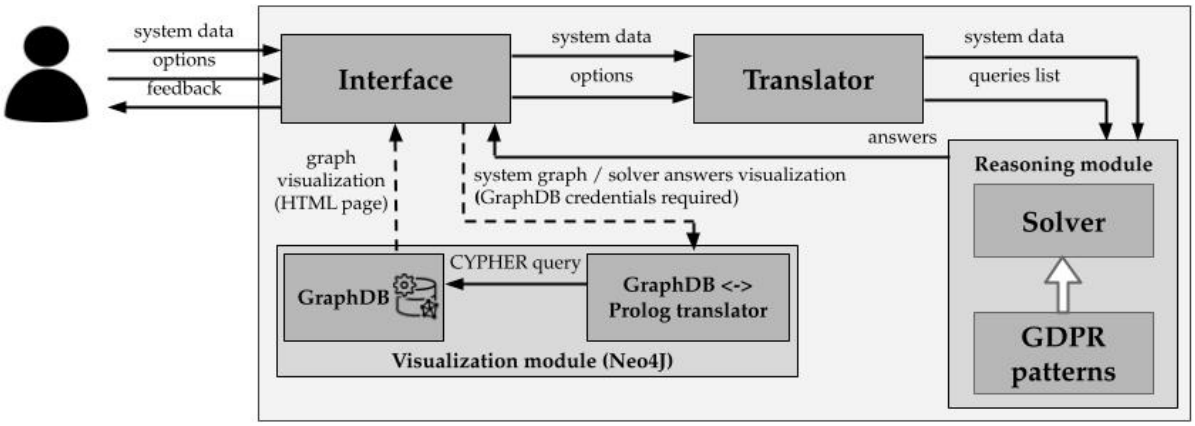

Figure 1 : Structure du prototype

- Le module Interface gère les entrées / sorties
- Le module Translator gère la traduction des demandes utilisateur en Prolog
- Le reasoning module envoie les requêtes au solveur Prolog en lien avec les patterns Prolog codant les principes du RGPD, récupère les résultats pour faire un retour utilisateur. Les résultats sont des instances d'une classe Issue stockées dans une liste de la classe Solver.
- Le module de visualisation gère le lien entre l'outil de GraphDB (Neo4j) et l'application. Il se charge de la conversion GraphDB ↔ graphe de provenance Prolog, et également de la génération du fichier HTML permettant l'affichage du graphe voulu

Cas d'utilisation type
Au lancement de l'application, l'utilisateur·ice doit fournir le fichier Prolog du graphe de provenance. Une autre option est d'importer le graphe de provenance stocké sous forme de GraphDB (outil Neo4J). Dans ce cas, l'utilisateur·ice devra fournir le lien, l'identifiant et le mot de passe de la base de donnée.

Figure 2 : Écran de sélection d'un graphe de provenance (Prolog)

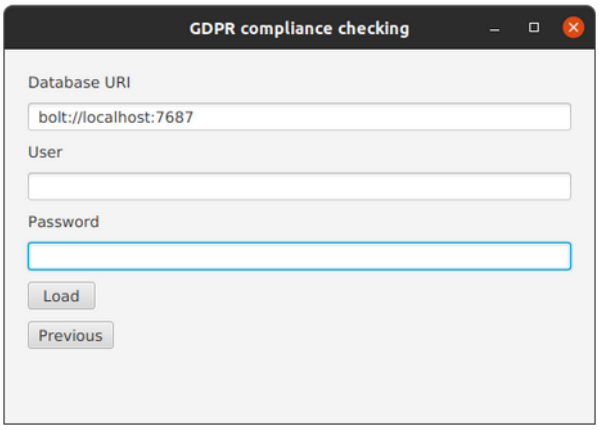

Figure 3 : Écran de connexion à une GraphDB Neo4j

Après le chargement du graphe de provenance, les utilisateur·ices, données et processus du système sont parsés et affichés dans l'application. Il est possible de les spécifier pour restreindre la recherche. Il est également possible de limiter la recherche à certains des 4 principes.

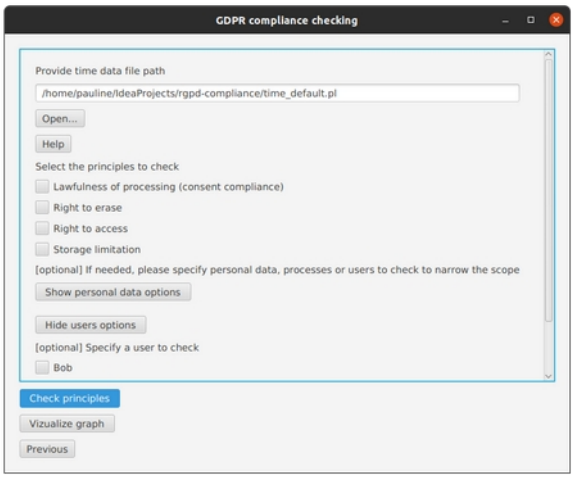

Figure 4 : Écran de sélection des paramètres de vérification

À ce stade, il est également possible d'afficher le graphe de provenance complet dans l'application. Si le graphe est stocké dans un fichier Prolog et pas dans une GraphDB, il faudra le charger dans la GraphDB afin de pouvoir l'afficher. L'utilisateur·ice est alors invité·e à se connecter, et reçoit un avertissement que la base de donnée actuellement stockée à cette adresse sera effacée et remplacée.

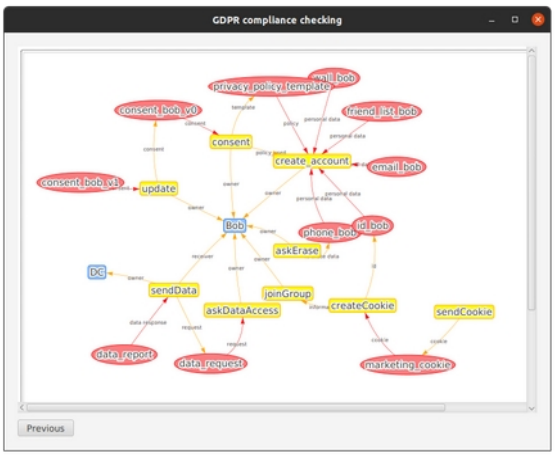

Figure 5 : Écran de visualisation du graphe complet

Depuis l'écran précédent (Fig. 4), on peut lancer la vérification. Les différents problèmes détectés seront récupérés du solveur Prolog et stockés dans des structures de données de type Issue. Chaque instance de cette classe se compose de :

- un type (le principe RGPD concerné)
- les instanciations de variables pour le pattern Prolog concerné
- un identifiant numérique unique permettant de la retrouver plus facilement

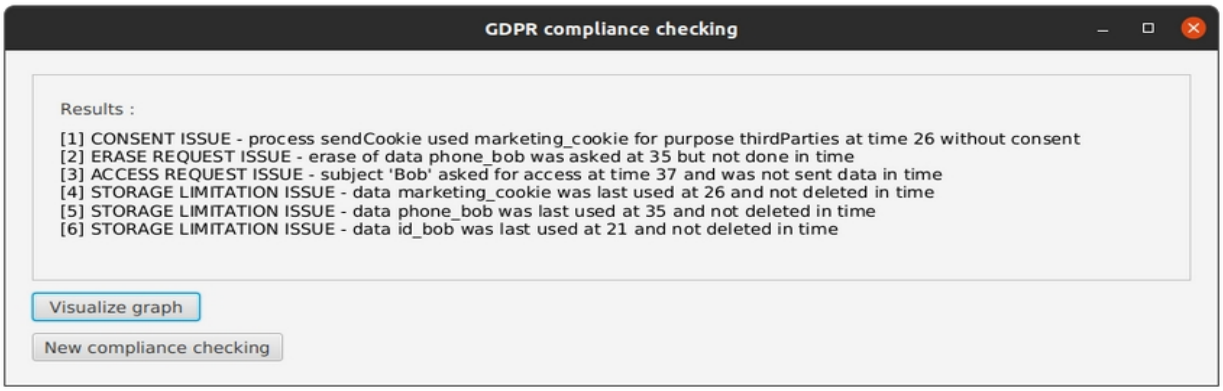

Figure 6 : Écran d'affichage des résultats

Par exemple, sur l'écran de résultats, on pourrait avoir la ligne suivante :

```
[14] CONSENT ISSUE - process sendMail used mail_user1_1 for purpose sendMail at time 11 without consent
```

Cela correspondra à une instance de Issue caractérisée par :

- son type (non-respect du principe de consentement)
- les instanciations de variables pour ce pattern Prolog, soit `P = sendMail`, `D = mail_user1_1`, `PU = sendMail`, `T = 11`
- l'identifiant numérique unique 14

Depuis l'écran de résultats (Fig. 6), on peut accéder à une fenêtre permettant de visualiser chacun des problèmes soulevés. Pour cela, de même que précédemment, le graphe de provenance devra être importé dans une GraphDB si ce n'est pas déjà le cas. On arrive ensuite à un affichage légèrement différent par rapport au graphe complet ; cette fois-ci, les différents problèmes sont listés sur la gauche de l'écran et on peut les sélectionner afin d'accéder à la portion de graphe correspondante. Pour cela, l'application réalise des requêtes à la GraphDB. Chaque Issue peut générer sa requête correspondante en fonction de ses paramètres et du principe auquel elle déroge.

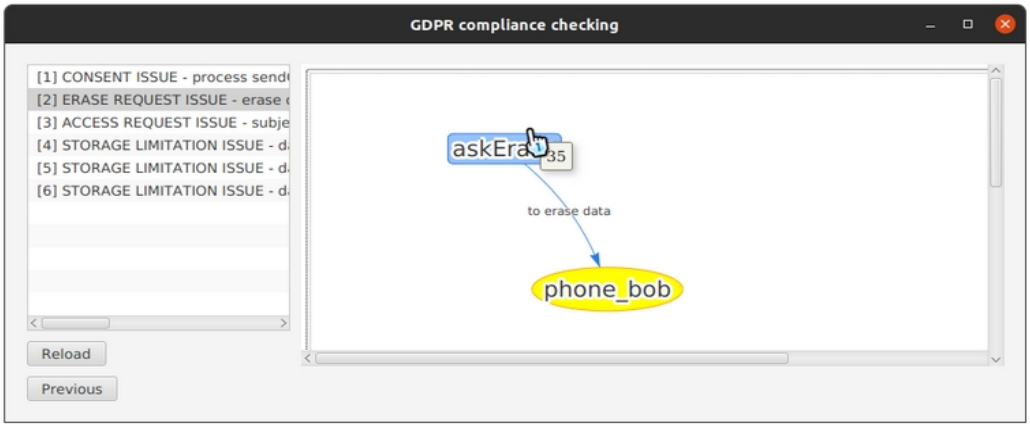

Figure 7 : Écran de visualisation des sous-graphes associés aux problèmes détectés

b. Parsers

Le lien avec les GraphDB se fait par la classe java Neo4jInterface. Elle est chargée :

- du stockage des informations de connexion
- de la transformation graphe Prolog ↔ graphe stocké dans la GraphDB
- de la génération du fichier HTML permettant la visualisation (au choix, le graphe complet ou un sous-graphe en fonction de la requête. Les requêtes seront détaillées dans la partie suivante)

→ Correction nécessaire : les informations d'identification sont actuellement stockées à plusieurs endroits du code, dont dans la classe Neo4jInterface par exemple.

Prolog → GraphDB

L'application passe systématiquement par le format GraphDB pour la visualisation d'un graphe de provenance. Un parser a donc été réalisé permettant de transformer un graphe de provenance Prolog en graphe stocké dans la base de données Neo4J.

La transformation est assez intuitive par rapport aux schémas fournis avec le sujet de TER.

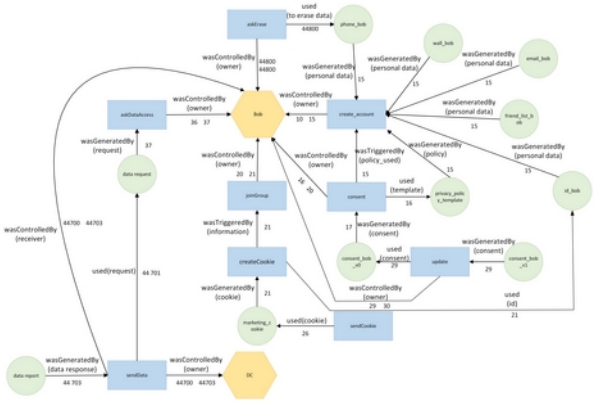

Figure 8 : Schéma d'un graphe de provenance fourni dans le TER (social network)

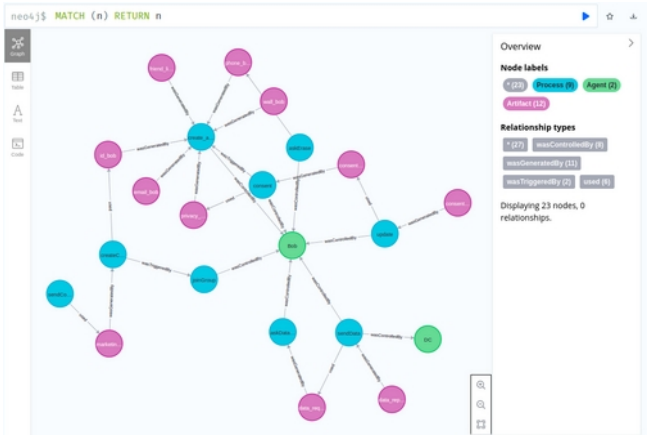

Figure 9 : Version GraphDB du graphe social network (depuis Neo4j browser)

Les nœuds sont les processus (`Process`), artefacts (`Artifact`) et agents (`Agent`), chacun identifiés par leur nom. Les relations `wasControlledBy`, `wasGeneratedBy`, `used`, `wasDerivedFrom`, `wasTriggeredBy` ont pour paramètres les informations affichées sur le schéma de gauche (temps, infos supplémentaires de contexte identifées dans la GraphDB par la propriété `ctx`).

Les nœuds représentant un consentement stockent les fins d'utilisations autorisées dans leurs propriétés. Par exemple, supposons que le graphe de provenance Prolog indique qu'un·e utilisateur·ice autorise les finalités suivantes :

```
purposes('consent_Alice_v1','phone_Alice',['thirdParties']).
purposes('consent_Alice_v1','mail_Alice',['analysis','thirdParties']).
purposes('consent_Alice_v2',_,['improvement','analysis']).
```

Le nœud représentant '`consent_Alice_v1`' aura pour propriétés :

- `phoneAlice : ['thirdParties']`

- `mail_Alice : [‘analysis’, ‘thirdParties’]`

Le nœud représentant ‘`consent_Alice_v2`’ aura pour propriété `_ : [‘improvement’, ‘analysis’]`.

Pendant le stage, nous avons eu des discussions autour du prédicat `action` (associant un processus à la finalité de son action), et de sa nécessité pour les processus n’utilisant pas de données personnelles. En effet, la façon dont sont codés les patterns Prolog nécessite la présence de prédicats ‘`action`’ pour chacun des processus du graphe (alors que dans les exemples consultés, la finalité associée portait souvent le même nom que le processus). Cette forme se traduit bien en GraphDB, étant donnée qu’un nœud de consentement aura simplement une propriété supplémentaire représentant l’action associée.

→ Correction nécessaire : Le générateur de graphes Prolog réalisé par le camarade pendant le TER donne le même nom aux processus de même type. Typiquement, on aura un unique processus `updateData` qui réalisera toutes les mises à jour de données. C’était le cas sur les exemples fournis, cependant sur de gros ensembles où les processus réalisent un plus grand nombre d’actions, on ne peut pas distinguer les processus dans le graphe. Cela peut poser problème sur la recherche de la donnée personnelle associée à un artefact. En effet, le pattern `isPersonal` vérifie si une donnée est dérivée (`wasDerivedFrom`) ou est elle même une donnée personnelle (identifiée par un prédicat `wasGeneratedBy(D,P,’personal data’, TG)`). Cependant, le prédicat `wasDerivedFrom` est codé par le pattern suivant :

```
wasDerivedFrom(D1,D2,R1,TG) :- wasGeneratedBy(D1,P,_R,TG) , used(P,D2,R1,_TU).
```

Prenons l’exemple du processus `updateData`. Soient les données `’mail_Alice_2’` et `‘phone_Bob_3’`. Ces données sont issues de mises à jour (respectivement la 2e version du mail d’Alice et la 3e version du téléphone de Bob). Or, elles sont toutes deux mises à jour par le processus `updateData`. Si le processus porte le même nom sur ces deux mises à jour de données, selon l’ordre des lignes dans le graphe, il se peut que `wasDerivedFrom(‘phone_Bob_3’,’mail_Alice_2’, R, TG)` soit évalué comme vrai. Cela pourrait conduire le solveur à identifier `’mail_Alice_2’` comme la donnée personnelle dont est issu `‘phone_Bob_3’`. Ce problème a été détecté tardivement lors de la génération de grands graphes, je n’ai donc pas eu le temps de le corriger.

Cela explique la présence d’un prédicat ‘`action`’ par processus ; le prédicat lie alors chaque nom (unique) de processus à sa fin d’utilisation associée. Il est à noter que les résultats renvoyés après analyse de la version GraphDB d’un graphe de provenance généré par cet outil peuvent différer par rapport à sa forme Prolog, cela est dû à l’ordre des lignes dans le fichier Prolog généré. Avec des noms de processus identiques tout au long du graphe de provenance, le solveur peut identifier la mauvaise donnée personnelle et ainsi vérifier le mauvais consentement. À priori, générer des graphes en mettant à jour les noms des processus afin d’en avoir un unique pour chaque utilisation devrait régler ce problème.

→ Correction nécessaire : Afin de gérer les fins d’utilisations auxquelles l’utilisateur·ice consent « par défaut » (`updateData`, `sendData`, consent, etc.), un nœud isolé est généré dans la GraphDB, portant le nom de ‘`mandatory_consent`’. Il liste les finalités auxquelles consent chaque utilisateur·ice du système en l’utilisant. Si la liste de fins d’utilisation est une liste « type » pour tous les systèmes, il est possible de l’ajouter en clair dans le fichier legal.pl (encodage Prolog du

principe de respect du consentement) afin de la faire sortir du graphe de provenance. Une autre option alternative serait de demander à l'utilisateur·ice de renseigner cette liste à part du graphe de provenance, de la même manière qu'on récupère les informations sur les limites de temps autorisées par le système.

GraphDB → Prolog

Pour effectuer une vérification des principes du RGPD, le graphe doit nécessairement être disponible sous forme Prolog. Dans le cas du chargement d'un graphe depuis une GraphDB, une copie sera écrite localement dans un fichier Prolog. Pour cela, un parser permet de réaliser la conversion (de même qu'en sens inverse, cette conversion est gérée par la classe Neo4jInterface).

Les points soulevés dans la partie Prolog → GraphDB s'appliquent également pour ce cas. En particulier, le consentement « obligatoire » sera extrait d'un nœud que l'on s'attend à voir nommé '`mandatory_consent`'. De plus, l'ordre de parcours de la GraphDB peut modifier l'ordre des lignes par rapport à la version du graphe de provenance issue du générateur, ce qui peut provoquer des différences de résultats en lien avec le problème de nommage des processus.

c. Requêtes CYPHER

Afin d'afficher le graphe (complet ou sous-portion), on réalise des requêtes à la GraphDB. Ici, on fait des requêtes utilisant le langage CYPHER (langage propriétaire associé à l'outil Neo4J). La syntaxe est semblable aux autres bases de données (telles que SQL) et permet de récupérer les nœuds et relations voulues.

Graphe complet

```
MATCH (n)-[r]->(m) RETURN *
```

Cette requête permet d'afficher l'ensemble des nœuds liés par des relations dans notre graphe. Les nœuds isolés ne sont pas retournés, mais nous n'en avons pas dans les graphes de provenance où il semble que les relations définissent les processus. Seul l'artefact '`mandatory_consent`' évoqué plus haut est absent de l'affichage

Respect du consentement

```
MATCH (p:Process {name: P})-[r:used {TU: TU}]->(d:Artifact {name: D}) RETURN
p,r,d
```

Cette requête affiche l'utilisation illicite de la donnée ainsi que le processus responsable.

→ Améliorations envisagées :

- dans le cas d'une utilisation non-conforme concernant une **donnée personnelle**, afficher l'endroit où la donnée a été générée
- dans le cas d'une utilisation non-conforme concernant une **donnée dérivée de donnée personnelle**, afficher le chemin vers la génération de la donnée personnelle
- afficher le consentement problématique

Respect des délais après demande d’accès

```
"MATCH (p:Process {action: 'askDataAccess'})-[r:wasControlledBy {TE: TE}]-
>(a:Agent {name: S}) RETURN p,r,a"
```

Cette requête affiche la demande d’accès non respectée.

→ Amélioration : si la demande a été satisfaite trop tardivement, afficher l’envoi de données.

Respect des délais après demande d’effacement

```
MATCH (p:Process {name: P, action: 'askErase'})-[r:used {TU: T}]->(d:Artifact
{name: D}) RETURN p,r,d
```

Cette requête affiche la demande d’effacement non respectée.

→ Amélioration : si la demande a été satisfaite trop tardivement, afficher l’effacement de données.

Respect des délais de stockage après la dernière utilisation

```
MATCH (p:Process)-[r:used {TU: TU}]->(d:Artifact {name: D}) RETURN p,r,d
```

Cette requête affiche la dernière utilisation de la donnée.

→ Amélioration : si la donnée a été effacée trop tardivement, afficher l’effacement de données.

Figure 10 : Affichage du problème [1] (consent)

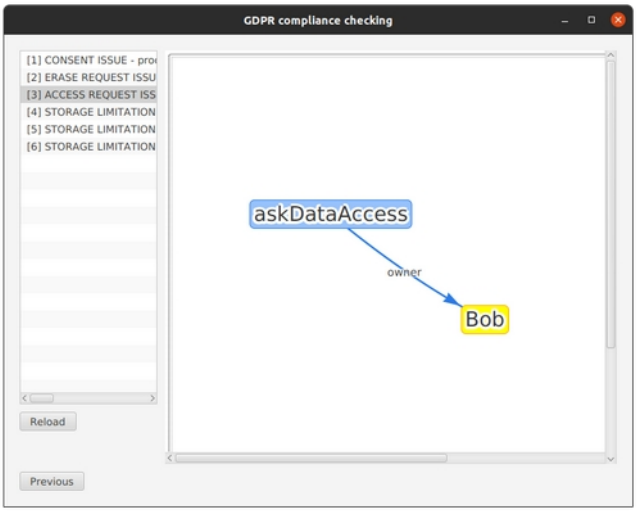

Figure 11 : Affichage de [3] (access)

Figure 12 : Affichage de [2] (erase)

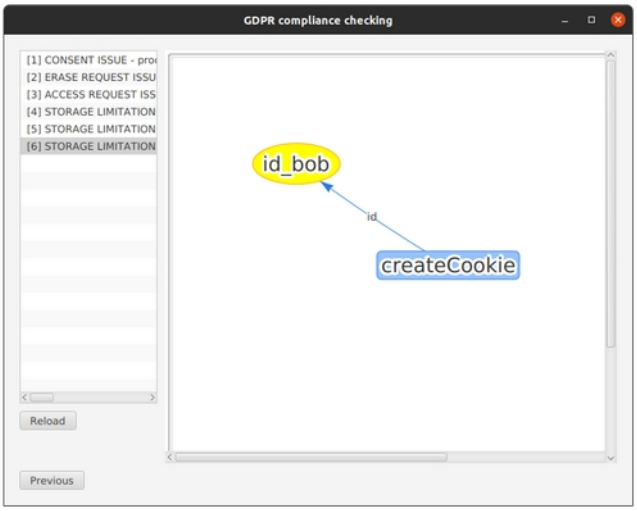

Figure 13 : Affichage de [6] (storage)

d. Affichage

L'affichage se fait par la création d'un fichier HTML via l'appel à la méthode `Neo4jInterface.buildVizHtmlFile(String query)`. On utilise NeoVis, outil propriétaire permettant l'affichage de GraphDB pour Neo4J. À chaque fois que l'on veut afficher un graphe, un fichier HTML est généré. Il se base sur le pattern `index_pattern.html`, qui est complété à la demande avec les informations de connexion et la requête CYPHER appropriée.

→ Correction nécessaire : détailler le label sur les transitions du graphe afin qu'il affiche le type (`wasGeneratedBy`, `used`, etc.), le(s) temps et les informations additionnelles. Pour l'instant, seul le temps est affiché au survol.

→ Correction nécessaire : les identifiants de connexion sont écrits directement dans le fichier HTML généré (en plus d'être stockés dans la classe Neo4jInterface).

2. Vérification

Utilisation d'une version légèrement modifiée du générateur de graphes afin de tester les fonctionnalités ajoutées depuis la version produite en TER. La version de base du générateur a été réalisée par Hichame PAPA.

Le code du générateur de graphe modifié, fourni dans le rendu, s'utilise de la manière suivante :
`java GraphGenerator <outputFileName> [systemType] [nbUsers] [factor]`

où :

- `outputFileName` est le nom du fichier généré en sortie (stocké dans `GraphGenerator/Output`)
- `systemType` est le type de système au choix parmi PUBLIC_SERVICE, SOCIAL_NETWORK ou WEBSTORE
- `nbUsers` est le nombre d'utilisateur·ices souhaité (à noter qu'il est possible de charger un fichier utilisateurs de son choix, contenant un nom d'utilisateur·ice par ligne, comme c'était le cas dans la version de départ)
- `factor` est un facteur de taille pour le graphe généré, plus ce nombre est grand et plus le graphe obtenu sera grand

Si les paramètres supplémentaires ne sont pas spécifiés, le graphe généré sera d'un type aléatoire (comme c'était le cas dans la version initiale), le nombre d'utilisateur·ices sera également aléatoire, et le facteur sera 1.

Les graphes obtenus sont fournis en annexe. Un tableau récapitulatif des temps de résolution mesurés pour différents graphes est également disponible sur le répertoire GitHub. Il liste les fichiers, leurs types, nombre d'utilisateur·ices et facteurs.

Ces tests ont permis de se rendre compte de plusieurs erreurs, mais aussi de réfléchir à des moyens d'améliorer la vitesse du temps de calcul. Par exemple, il semble que le prédicat `isPersonal` soit très coûteux étant donné qu'il regarde l'ensemble des données personnelles pour vérifier si la donnée courante en est dérivée. Il se peut que le calcul puisse être accéléré en modifiant l'ordre des prédicats des patterns RGPD afin que le solveur Prolog puisse faire les « coupures » plus vite.

→ Améliorations : vérifier l'ordre des prédicats notamment dans `isPersonal`.

3. Récapitulatif des tâches du stage

- (Préparation d'un support de présentation)
- (Participation à la rédaction d'un extended abstract)
- Traduction de l'interface en anglais
- Ajout d'un module gérant la conversion entre GraphDB (Neo4j) et Prolog
- Ajout d'une fonctionnalité permettant de visualiser le graphe complet ou un sous-graphe centré sur un événement ne respectant pas le RGPD (en passant par les outils Neo4j)
- Modification du générateur de graphes
- Génération d'un set de graphes avec plus d'utilisateur·ices
- Ajout d'une structure de donnée pour représenter les problèmes de non-respect du RGPD identifiés par le solveur
- Modification de structure de la classe Solver qui récupère maintenant les affectations de variable du solveur Prolog (dans la version précédente, le solveur Prolog écrivait la réponse directement dans la console, et le flux était redirigé pour affichage dans l'application). Cela permet de manipuler les résultats obtenus

4. Récapitulatif des améliorations / corrections envisagées

- Corriger la façon dont sont stockées les informations de connexion (actuellement en clair dans le fichier HTML d'affichage et en variables dans la classe Neo4jInterface)
- Modifier le générateur de graphes pour que les processus aient des noms uniques
- Modifier la façon dont est stocké / récupéré le consentement « obligatoire » (le demander à l'utilisateur·ice de l'application ? l'ajouter en clair dans le fichier `legal.pl` ?)
- Améliorer les affichages de sous-graphes pour qu'ils soient plus complets (et qu'ils intègrent plus d'éléments du graphe)
- Détailler les informations affichées sur le graphe (notamment sur les transitions)
- Vérifier l'ordre des prédicats Prolog des patterns RGPD pour améliorer les performances

5. Annexe

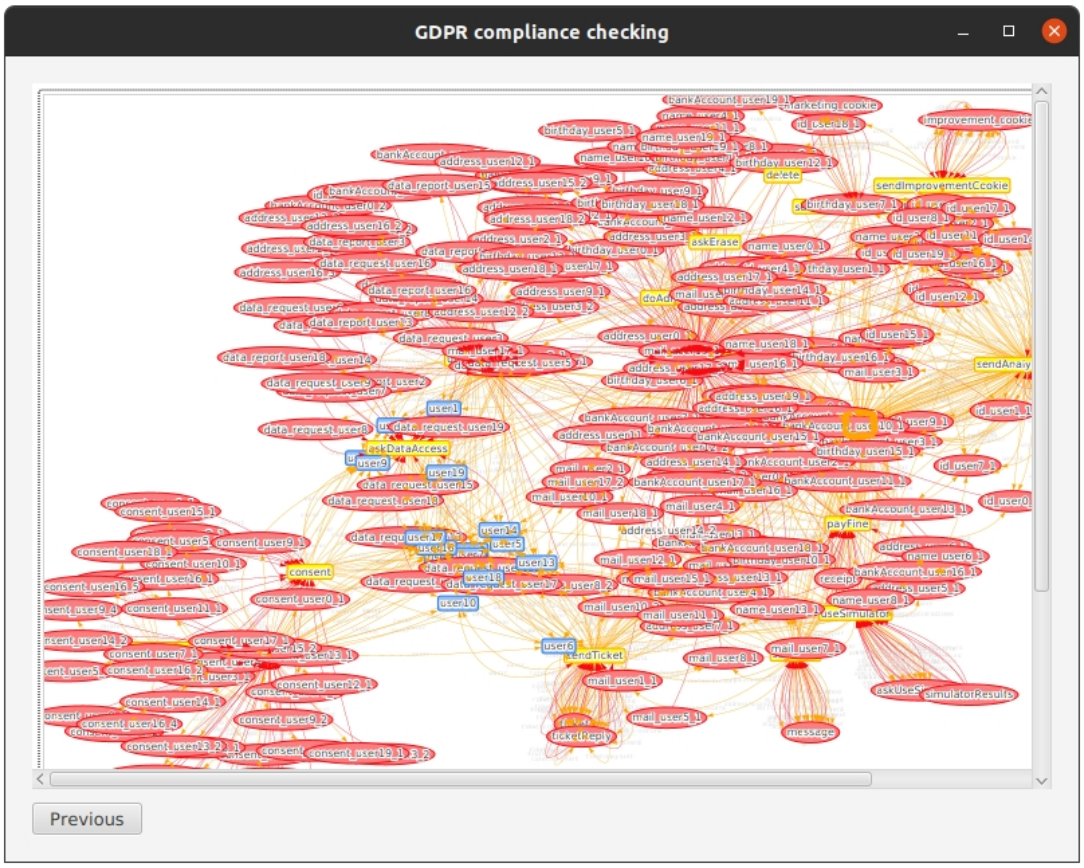

Figure 14 : Visualisation d’un graphe à 20 utilisateur·ices (20_5_PS.pl)

Remarque : Les figures de ce rapport sont disponibles sur le répertoire GitHub du projet.