Conception d'un Langage d'Interrogation des Traces Accessible à des Non-Informaticiens

Bryan Kong Win Chang, Nathalie Guin, Marie Lefevre, Pierre-Antoine Champin

Université de Lyon, CNRS
Université Lyon 1, LIRIS, UMR5205, F-69622, France
{Bryan.Kong-Win-Chang,
Nathalie.Guin,Marie.Lefevre,Pierre-Antoine.Champin}@liris.cnrs.fr

Résumé Les traces d'un logiciel sont une source d'informations sur l'activité d'un utilisateur lors de l'utilisation d'un système. Permettre à l'auteur d'un Environnement Informatique pour l'Apprentissage Humain (*EIAH*) (ou un autre utilisateur) d'accéder aux informations contenues dans ces traces d'activité afin d'alimenter un profil d'apprenant reste un problème récurrent dans les conceptions des EIAH, souvent traité *via* des solutions ad-hoc cadrées par un expert informaticien. Au sein de l'équipe SILEX a été développé une plateforme basé sur du RDF, le kTBS, kernel for Trace-Based-System, qui implémente la notion de Système à Base de Trace (*SBT*), notion définissant une manière de stocker et utiliser les traces d'interactions. Dans ce mémoire, nous présentons un langage inspiré du langage naturel pour permettre à un non-informaticien de définir son propre indicateur. Nous présentons ensuite une interface graphique permettant à l'utilisateur novice d'appréhender la logique du langage et d'en assimiler le fonctionnement plus facilement.

Abstract. Traces can be a useful source of information for better understanding of the user's behavior on a given system. The lack of availability of this information for a basic user (i.e. not an expert in computer science) is a main issue for the development of adaptive E-learning systems. In this report, we introduce a language inspired by Controlled Natural Language that aims at easing the querying of kTBS, an RDF implementation of the Trace Based System meta-model. That language provides a way to create indicators using formatted french natural language. We also describe two basic interfaces which should ease the understanding of the language for a novice user.

Keywords: IA, EIAH, Système à Base de Trace, Traces d'Activités, Indicateurs, Langage.

1 Introduction

Les traces d'un logiciel sont une source d'informations et de connaissances sur l'activité d'un utilisateur *via* l'utilisation du logiciel. Permettre à l'auteur d'un

EIAH de pouvoir exploiter ces traces pour récupérer les informations qu'elles contiennent pour alimenter un profil d'apprenant reste un problème récurrent dans l'ingénierie des EIAH, souvent traité *via* des solutions ad-hoc cadrées par un expert informaticien. Au sein de l'équipe SILEX a été développée une plateforme basée sur du RDF, le kTBS, kernel for Trace-Based-System, qui implémente la notion de Systéme à Base de Trace (*SBT*), notion définissant une manière de stocker et utiliser les traces d'interactions.

Avant de pouvoir parler des traces d'interactions et de l'implémentation d'un langage du kTBS, il convient de poser certaines bases dans cette introduction, afin d'expliciter le contexte menant à la volonté de construire ce langage. La première est la notion d'indicateur et de profil d'apprenant, la seconde la notion de système de trace et de l'implémentation kTBS.

1.1 Intérêt pour les Traces

La notion de *trace* est généralement admise comme le résultat d'une activité. Dans le cadre d'un logiciel informatique, on la définit dans un premier temps simplement comme ce que le logiciel produit. Cela va de la sortie écran à toutes autres informations, dont celles non transmises directement à l'utilisateur. On retrouve ainsi par exemple dans la définition large des *traces* les logs d'un programme, ou encore les préférences d'un utilisateur pour le paramétrage de son interface visuelle.

Ce qui reste, ces traces, le témoin de l'activité qui a eu lieu, étaient alors souvent inutilisés ou seulement destinées à l'utilisation par des experts du logiciel afin de mieux comprendre un comportement anormal (les rapports d'erreurs, par exemple, utilisent souvent ces logs).

Dans les logiciels éducatifs, cependant, on s'intéressa rapidement à ces traces dans le but de fournir à l'apprenant une personnalisation de son apprentissage grâce au stockage dans ses traces d'informations considérées comme pertinentes par le concepteur. On retrouve alors des rapports sur des informations brutes, comme par exemple sur le contenu vu par l'utilisateur ou sur les résultats d'un exercice, ou des informations calculées à partir de ces données.

Ces informations peuvent ensuite être utilisées directement par le système en changeant son comportement ou indirectement rapportant les informations à un enseignant ou à un apprenant qui se charge ensuite de l'adaptation. Ces informations récupérées restent extrêmement disparates et les manières de les calculer sont variées . Un état de l'art de 2004 [1] sépare les indicateurs en sept catégories, chaque catégorie représentant un ensemble allant de 5 à 8 sous-catégories selon l'information récupérée par l'indicateur. On retrouve ainsi des méthodes "simples", qui infèrent l'acquisition d'une connaissance par un apprenant *via* une simple évaluation, tandis que d'autres essayeront d'appliquer des méthodes statistiques pour s'assurer que l'évaluation ne résulte pas d'un coup de chance de l'apprenant à un exercice particulier.

L'ensemble de ces informations, brutes ou calculées, pour un apprenant donné, sont référées sous le nom de *profil d'apprenant*, tandis qu'une donnée calculée est généralement nommée *indicateur* [2].

1.2 Formalisation des indicateurs

De 2004 à 2008 se déroule le projet Kaléidoscope, ou Réseau Européen d'excellence Kaléidoscope, qui s'intéresse à faire collaborer les acteurs européens sur la scène du E-learning. De ce grand projet collaboratif est né plusieurs actions. L'une d'entre elle est le projet DPULS, Design Pattern for collecting and analysing Usage of Learning Systems. Ce projet mènera notamment à une formalisation de la notion d'indicateur à partir d'une étude complète sur une centaine d'indicateurs différents [1][2]. Cette définition très simple pose l'indicateur comme une variable mathématique indiquant la qualité de quelque chose, ce quelque chose englobant un ensemble d'activités entre l'apprenant et le système ou entre plusieurs apprenants *via* un système. Cette définition est accompagnée d'un pattern définissant ce qu'il est nécessaire de définir pour créer un indicateur. Ce pattern, composé entre autre d'un descriptif textuel de l'indicateur et d'une formule mathématique permettant d'en calculer la valeur, servira de base aux projets liés à la création d'indicateurs par des non-informaticiens [3][4][5], dont on reparlera dans la deuxième partie de ce document.

1.3 Les Systèmes à Base de Traces

La notion de Système à Base de Traces Modélisées nait dans ce contexte de vouloir comprendre l'activité de l'utilisateur pour lui apporter un retour sur lui même ou sur les autres. On peut situer le départ des travaux ayant mené aux Systèmes à Base de Traces en 2003, avec l'approche MUSETTE [6], qui adapte le Raisonnement à Partir de Cas (RàPC) pour y ajouter une notion temporelle *via* les contextes. Il s'agit de prendre en compte l'évolution de la façon dont doivent être considérés les cas selon un contexte temporel donné. Cependant, les limites du modèle mènent à énoncer la nécessité de changer plus drastiquement le modèle pour un Raisonnement à Partir de Traces [7]. L'utilisation des traces comme base amènent à la nécessité de travailler sur un formalisme pour ces traces. *Via* un parallèle avec les Systèmes à Base de Connaissances est alors proposé en 2009 une première version du Système à Base de Traces [8]. Le métamodèle proposé accorde une importance toute particulière à la notion de temps. La trace est alors définie comme ensemble d'observé situé dans le temps. Ces observés, des *observed elements*, sont nommés par contraction des obsels. Il y est également présenté les principaux algorithmes nécéssaire à l'implémentation d'un tel système. Le rapport de thèse du même auteur, en 2011 [9], présente d'ailleurs une étude approfondie des langages les plus intéressants pour l'implémentation d'un *SBT*, en pointant au passage un certain nombre de notions non prises en charge basiquement par les langages étudiés. Il est principalement mis en avant le RDF, bien que des complications possibles soient évoquées pour l'analyse des traces en temps réels dans le cas d'un flux important (Partie 5, Discussion du document [9]).

1.4 Kernel for Trace Based System

Le Kernel for Trace Based System (kTBS) est à considérer comme une implémentation du modèle *SBT* présenté par L. S. Settouti dans sa thèse [9]. Cette plateforme, comme son nom l'indique, se veux comme une base générale pour les Systèmes à Base de Traces. Utilisant les technologies du web, la plateforme stocke les données en RDF. L'accès se fait donc principalement *via* le SPARQL, langage fortement inspirée du SQL, servant pour l'interrogation des bases RDF.

Cependant, malgré les avantages l'utilisation de SPARQL pour la modélisation des traces [9], le kTBS présente les mêmes problèmes que ceux soulevé en web-sémantique sur sa difficulté d'utilisation par le grand public. Ce dernier requiert la compréhension de la structure de graphe RDF pour pouvoir exprimer les conditions sur les *obsels* et du SQL pour sa structure générale de requête. Les utilisateurs habitués à la structure SELECT-FROM-WHERE auront ainsi beaucoup plus de facilités à utiliser le SPARQL.

1.5 Problématique

Le sujet "Conception d'un langage d'interrogation des traces destinées à un non informaticien" est réalisé dans le cadre du projet Connaissance Ouverte à Tous (*COAT*) du CNRS[1]. Dans le cadre d'une de ses premières applications réelles, le kTBS est utilisée dans la récupération de traces d'utilisateurs d'un *MOOC* (Massive Open Online Courses), FOVEA. Les *MOOC*, catégorie particulière d'Environnement Informatique pour l'Apprentissage Humain (*EIAH*), sont aussi confrontés aux problèmes rencontrés par l'apprentissage à distance. Un des moyens pressenti pour réduire les problèmes d'abandon en cours de *MOOC* est la personnalisation du parcours de l'apprenant. À défaut de pouvoir suivre chaque apprenant, l'analyse des usages *via* les traces devrait permettre à un auteur de MOOC de définir des stratégies d'adaptation selon des conditions sur le profil d'apprenant. Toutefois, si l'auteur d'un MOOC est un expert dans son domaine, il n'est pas forcément un expert en informatique.

De fait, dans le cas d'un utilisateur non-informaticien, comment définir un langage lui permettant de requêter les traces des utilisateurs stockées dans un kTBS sans lui imposer l'utilisation du SPARQL ?

Cependant, la notion de non-informaticien est à relativiser. On ne parle pas ici de quelqu'un complètement étranger au monde de l'informatique. Il s'agit ici de ne présupposer aucune compétence de programmation. On pourra supposer d'autres connaissances, comme la capacité de reformuler sa demande et les notions de base du *SBT*, soit la notion d'obsel et de traces.

Le langage proposé ayant pour but l'interrogation de traces, il conviendra aussi de s'appuyer sur les besoins exprimés lors de la définition des *SBT* et n'étant pas couverts par des opérations de base du langage SPARQL. Ces opérations sont principalement temporelles, du fait que pour le RDF, le temps est une

1. http://liris.cnrs.fr/coatcnrs/wiki/doku.php

donnée de même importance qu'une autre. Le résultat de la requête ayant pour but l'extraction d'éléments de ces traces, il devra également permettre des calculs sur ces éléments, afin de pouvoir permettre la réalisation d'indicateurs, usage principal des utilisateurs ciblés.

Dans ce mémoire, nous présenterons d'abord différentes solutions abordées pour accéder à ces traces d'interaction. Nous continuerons ensuite dans la section 3 par la présentation du langage proposé avec divers exemples d'utilisation et les choix et compromis faits pour faciliter son utilisation. Nous présenterons ensuite dans la section 4 un implémentation de ce dernier dans le cadre d'une interface html et d'un parser codé en javascript. Nous terminerons par une dernière partie sur les évaluations et les perspectives du travail.

2 État de l'Art

Dans cette partie, nous présenterons en premier lieu les travaux sur les outils ayant pour but de faciliter l'ingénierie des indicateurs en EIAH. Cette partie inclura des travaux utilisant la notion de SBT [8] et d'autres s'appuyant plus sur la façon d'instancier les patterns d'indicateurs définis par le groupe Kaléidoscope [2]. La deuxième partie présentera des travaux en lien avec le web sémantique et s'appuiera sur des exemples des approches proposées pour rendre le web sémantique accessible à tous.

2.1 Indicateurs

La définition par le groupe Kaléidoscope d'un indicateur [2] est basée sur une analyse comparée de l'ensemble des indicateurs utilisé dans les EIAH (plus de 40 indicateurs sont ainsi répertoriés). L'analyse a permis de définir un modèle générique pour les indicateurs. Cette base solide, proposant l'instanciation de l'ensemble des indicateurs analysés, repose sur un cadre générique et une spéficiation permettant de classer l'indicateur selon son type et son usage. Il est par exemple explicité les indicateurs de collaboration et les types de collaboration. Basé sur la généricité de ce modèle, on retrouve dans la littérature un certain nombre de propositions sur un moyen générique de permettre à un auteur d'EIAH de créer ses propres indicateurs.

EM-AGIIR (Environnement Multi-AGent de supervIsion à base d'Indicateurs Réutilisés) [5] propose une première approche afin de faciliter l'ingénierie des indicateurs pour les EIAH. L'environnement se fonde sur une base de donnée traditionnelle afin de stocker les traces. La notion de base de donnée traditionnelle est à opposer aux système représentant le raisonnement d'un expert (ou système à base de règles) et aux systèmes destinées spécifiquement à la gestion de flux. Dans EM-AGIIR, la base SQL est couplée par un expert à un nombre donné d'informations qui nous intéressent. Il s'agit là d'extraire des *traces brutes* les *traces primaires*. Cette première phase est en fait une première étape de sélection de ce qui est intéressant de récupérer dans l'ensemble des traces. Cette phase

est souvent déjà effectuée dans de nombreux EIAH existants, où le contenu de la trace est prévu par avance par ce que l'on veut connaitre des interactions de l'apprenant et du système. Une des originalités de l'approche est de proposer un ensemble d'agents qui surveillent à intervalle régulier les modifications de la base de traces afin de mettre à jour l'indicateur qui lui est lié. L'interface utilisateur pour la création d'un indicateur de l'environnement EM-AGIIR (cf. Fig.1) est graphique.

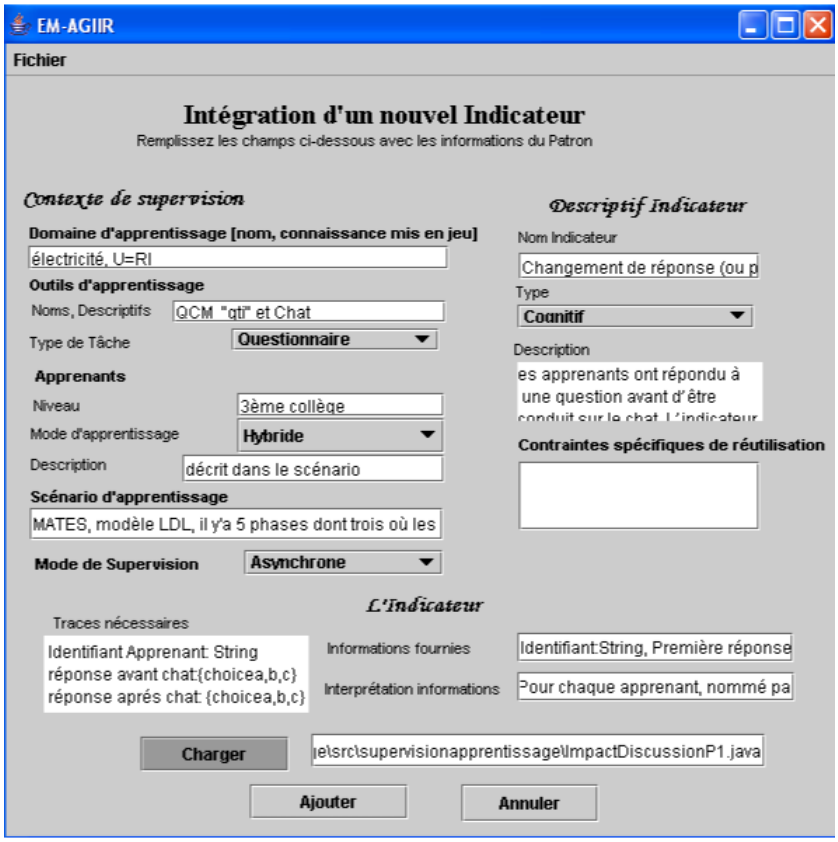

Fig. 1. EM-AGIIR : Interface d'intégration d'un nouvel indicateur [5]

Via la proposition d'un formulaire simple d'utilisation, l'interface organise la page pour aider la réflexion de l'auteur d'indicateurs. Par le choix de la disposition des éléments de la fenêtre, l'utilisateur remplissant le formulaire est appelé à progressivement renseigner des informations de plus en plus bas-niveau. Ce choix permet d'amener ce dernier à réfléchir à la création de son indicateur, en le guidant de la description générale de son indicateur au calcul formel de ce dernier. Il est intéressant de noter que le champ "informations fournies" ne

contient pas directement des éléments de la trace, mais des noms de variables à discrétion de l'auteur de l'indicateur. Une autre interface, proposée par la suite, demande à l'utilisateur de faire le lien entre les variables indiquées dans ce champ et les données des traces (présentées sous forme de liste). Un autre avantage important de EM-AGIIR réside dans le système de capitalisation des indicateurs *via* la mise en place d'un catalogue des indicateurs. Ce catalogue permet à un autre auteur de s'inspirer d'un indicateur existant ou de le récupérer s'il répond directement à ses besoins.

Les traces utilisées ici se basent sur la notion de trace définie dans le cadre d'UTL2 [10]. Dans le cadre de ce méta-modèle générique pour l'ingénierie de traces, la définition de trace est beaucoup plus large que dans la définition du *SBT*. Ainsi, EM-AGIIR ne s'intéresse pas particulièrement à l'évolution temporelle des traces. De plus, dans le cadre où l'utilisateur final est un non-informaticien, EM-AGIIR n'est pas directement utilisable du fait qu'il demande à l'utilisateur de fournir le code de l'agent sous forme d'un modèle complété.

GINDIC (Génération d'INDICateurs [3]) est défini comme un outil pour l'élaboration d'indicateurs. Il est ciblé dans ce travail un type d'indicateurs particulier au sens de [2], puisqu'on s'intéresse particulièrement aux indicateurs de collaboration. L'outil en lui-même est indépendant de la plateforme produisant les traces à observer. L'interface de création de l'indicateur affiche de fortes similarités avec l'approche précédente. Cependant, là où EM-AGIIR choisit un système à base d'agents pour surveiller la base en informant le système des changements de données pour mettre à jour les indicateurs, GINDIC utilise un système à base de règles. L'avantage du premier est un gain de temps sur les bases se modifiant souvent (ce qu'on peut constater dans le cas du kTBS). Le second gagne l'avantage de ne pas avoir besoin de demander à un auteur du code afin d'utiliser l'outil. Pour ce qui est de l'interface de création des indicateurs, on retrouve une présentation extrêmement similaire à celle de l'exemple précédent (cf. Fig.2). L'élément numéroté 1 de l'interface est la décomposition de la création d'indicateurs en étapes distinctes. Chaque élément de 1 donnant accès à un formulaire dans la zone 2. L'étape de définition demande à l'utilisateur des informations générales sur l'indicateur, et est très similaire à la partie haute de l'interface de EM-AGIIR. Il y a deux avantages à cette approche. Les indicateurs collaboratifs ont pour objectif premier de comparer une activité, ou le nombre d'activités, vis-à-vis d'une autre variable. Cette autre variable peut être une simple donnée, comme le temps, ou encore l'indicateur d'une autre personne ou d'un groupe de personnes. Ils se prêtent ainsi parfaitement bien à une visualisation graphique en deux dimensions.

La présence d'une visualisation de l'indicateur sur les données sélectionnées est cité comme premier moyen d'évaluation de l'indicateur. Même si cela ne valide pas mathématiquement l'indicateur, l'auteur peut en vérifier la cohérence globale et ainsi être plus confiant dans les résultats fournis par ce dernier. L'autre

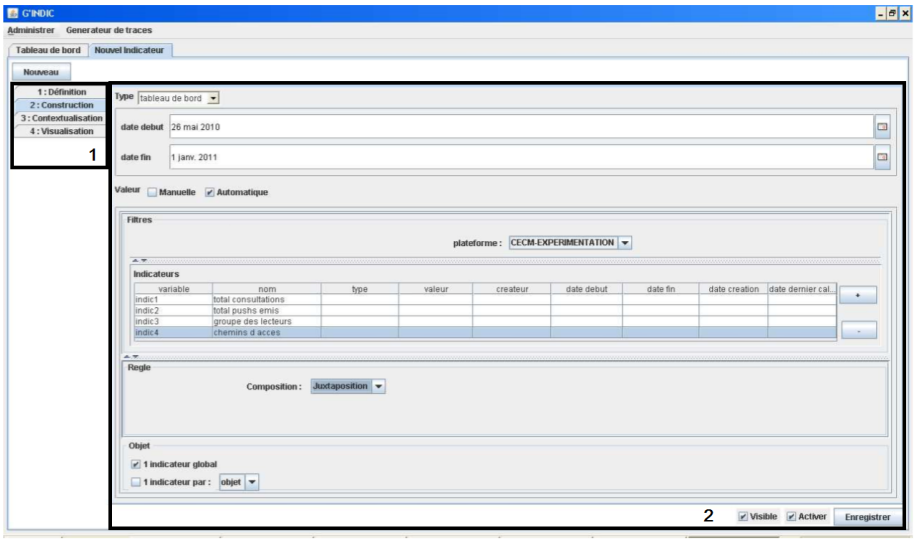

Fig. 2. Interface Graphique de GINDIC [3]

avantage de cette approche est l'ajout de variables temporelles permettant de créer un premier filtre sur la période temporelle qui intéresse l'utilisateur, l'aspect temporel étant un point important pour l'exploitation des traces dans les SBT. De plus, comme pour EM-AGIIR, on retrouve dans GINDIC un catalogue se prêtant à la réutilisation des indicateurs pour d'autres et par d'autres.

Il faut toutefois nuancer l'intérêt pour notre sujet par la limitation posée par le choix de considérer uniquement les indicateurs de collaboration, la visualisation proposant principalement des graphes exprimant une donnée par rapport à une autre, là ou l'indicateur pourrait être une simple valeur unique, ou même un simple booléen, et non une comparaison graphique de sa valeur sur une intervalle de temps.

SBT-IM. Contrairement aux deux précédents, le SBT-IM est développé directement en suivant la définition du SBT. De son nom complet Système à Base de Traces pour le calcul d'Indicateurs sur la plateforme Moodle, il s'agit d'une architecture particulière visant en premier lieu les indicateurs d'activité collaboratifs et individuels dans les activités de la plateforme collaborative Moodle [11].

L'outil s'est développé en parallèle du *SBT* en se basant sur sa définition et collaboration avec l'auteur du *SBT* [20]. De plus, il reprend les mêmes avantages (sauf la généricité de la platerforme étudiée) que les deux autres outils présentés ci-dessus. Il permet de choisir une visualisation et présente à l'auteur un système de renseignement d'informations progressif, des informations générales aux informations détaillées sur le calcul de l'indicateur en cours de définition.

Cependant, l'auteur des travaux se penche sur la difficulté qu'a un auteur à savoir ce qu'il peut calculer, et des informations mises à sa disposition. Il est donc mis en avant un moyen de pouvoir consulter et parcourir les traces durant la réalisation du dit indicateur. Cependant, si l'idée est bonne, le parcours des traces se fait *via* une suite de tableau affichant les donnée filtrées par opérations successives. L'ensemble des vues générées s'accumule assez rapidement et nécessite de connaitre ce que fourni la trace, du fait de la visualisation unique en tableaux de 4 colonnes. Les traces Moodle sont assez simples, n'ayant que 4 éléments, mais on peut se demander si cette approche est toujours viable pour des traces beaucoup plus compliquées, pour des utilisateurs moins experts que lors de son expérimentation [4] avec des étudiants en informatique en première année de Master. Des travaux en continuité de ces publications sont d'ailleurs en cours pour améliorer les défauts d'ergonomie et pour tenter de généraliser l'approche à d'autres traces que celles de Moodle, pour arriver à un *SBT-IX*, ou 'x' serait pour indiquer la généricité par rapport à la plateforme.

2.2 Les Langages pour le Web-Sémantique

Le web dans la globalité continue à prendre de l'ampleur tant dans son nombre d'utilisateurs que du nombre de données à exploiter. Le nombre de données RDF également. Dans le but d'interroger une architecture basée sur le RDF, il est pertinent de se pencher sur les différentes techniques utilisées pour faciliter l'accès de ces données à ce qui est appelé dans les articles "l'utilisateur lambda" du web. Abstraire le SPARQL est en effet une problématique qui a déjà vu de nombreuses propositions et adaptations plus ou moins réussies, et fait l'objet de plusieurs concours récompensant les outils de recommandation et de recherche permettant à l'utilisateur d'obtenir le plus facilement l'information qu'il veut (voir les concours QALD open challenge [19]).

Moteurs de Recherches Les moteurs de recherches sont certainement les outils les plus utilisés dans le cadre de la recherche d'informations sur le web. Leur utilisation, ancrée dans les tout débuts du web, propose un ensemble d'outils connus par l'ensemble des utilisateurs du web par leur grande popularisation. L'utilisation intuitive fonctionne par la saisie d'un certain nombre de mots liés à la recherche, que chaque moteur interprétera à sa manière pour en sortir une liste de suggestions que l'utilisateur final est libre de choisir. Ces processus vont du plus simple (simple présence des mots clefs dans le code des pages proposées) à des ontologies plus complexes de synonymes et autres contraintes sémantiques automatiquement trouvées ou bien manuellement imposées. Bien que non adapté à l'utilisation dans le cadre de la formulation d'indicateurs, en cherchant à satisfaire le besoin de l'utilisateur *via* une présentation globale de ressources qui pourraient l'intéresser, et ne permettant pas de calculer une donnée issue de cette recherche, il est intéressant de noter que ce genre d'approche (recherche par mot clefs) est celle qui à l'heure actuelle donne les meilleurs résultats dans les concours sus-nommés.

Pattern d'Utilisation L'approche consiste à considérer un ensemble de requêtes que l'utilisateur final sera potentiellement amené à vouloir réaliser sur une base de donnée donnée. Par exemple, dans SWIP [12], le principe est d'analyser le langage naturel pour obtenir les requêtes SPARQL. Ils considèrent la création manuelle d'un ensemble de patterns suffisant pour répondre aux besoins de l'utilisateur. Le processus SWIP, Semantic Web Interface using Pattern, passe du langage naturel à une liste de patterns de requêtes en trois étapes. La première étape est la récupération de la saisie utilisateur pour la transformer en requête d'un langage nommé le "langage pivot", qui se situe à mi-chemin entre les requêtes SPARQL et le langage naturel, et est considéré comme sémantiquement compréhensible par l'utilisateur. Elle se présente en une grammaire simple consistant en un ensemble de mots clefs et de points virgules séparant les conditions sur le contenu des éléments recherchés. Le point virgule, qui n'est pas sans rappeler la notion de variable SPARQL, est rajouté sur les éléments spécifiques qu'on veut récupérer. À partir de cette transformation, SWIP sélectionne un ensemble de patterns proche à l'aide d'un indice de confiance basé sur une ontologie de rapprochement des mots. L'ensemble des patterns les plus proches sont alors instanciés et proposés à l'utilisateur, qui peut alors sélectionner celui qui correspond le mieux à ses besoins. L'approche donne de bons résultats sur son implémentation pour MusicBrainz [13] dans le cadre du concours QALD 3, mais si les résultats sont plus (ou du moins aussi) pertinent que les approches par mots clefs, l'approche éprouve de grosses difficultés si le contenu des pages est plus irrégulier et à beaucoup plus de mal à s'adpater. Si la demande est hors des familles de patterns définis, l'approche n'arrive généralement pas à proposer quelque chose de similaire acceptable pour l'utilisateur. Les principes de l'approche sont indépendant du domaine d'application, mais l'ensemble des patterns à définir est un travail d'ingénierie supplémentaire lourd pour chaque nouveau domaine d'application.

Un Langage Naturel Controlé : SQUALL. Il s'agit ici de restreindre le langage naturel pour en faire un langage utilisable sans ambiguïté. Les règles imposées peuvent être des interdictions de synonymes, ou des règles invitant généralement à supprimer des ambuiguïtés sémantiques. Les approches de ce genre de langage sont beaucoup plus étudiées en anglais, où sont développés depuis longtemps des analyseurs sémantiques du langage (Montague 1973 [14]). SQUALL [15] est un langage, qui s'inspire de la théorie de Montague "Universal Grammar", théorie appuyant la possibilité de considérer les langages naturels et machines de la même manière [14]. Sa théorie explique cette possibilité *via* la mise en application de procédés algorithmiques simples pour passer d'un langage naturel à des langages formels simples, ayant menés aux parsers de Montague. SQUALL utilise ainsi cette base pour traduire des questions posées en anglais sur des données de DBpedia. Il explique également dans son article de 2013 [16] les difficultés rencontrées lors du passage au SPARQL et montre que son langage est aussi expressif que le SPARQL. Les résultats au concours QALD sont bons, avec un bon rappel sur les requêtes utilisées lors du concours. Un exemple de requête

du langage est par exemple : "What is the deathDate of res:Michael Jackson ?", qui permet de récupérer la date de mort du dit chanteur. On constatera ainsi que le langage reste très proche du langage naturel. Dans le cadre d'une utilisation *via* le kTBS, le langage SQUALL propose un langage indépendant de tout domaine, en se calquant plus sur le côté triplet "sujet prédication objet" du SPARQL lors de la traduction en langage formel. Cependant, orienté par cela, il n'est pas non plus considéré le temps ou la date des informations de manière particulière. Le "When" est ainsi absent du langage. Cela reste cependant un des langages les plus proches de la formulation utilisateur.

Samotrace Dans le cadre de l'accès aux traces, il nous semble par ailleurs intéressant de citer ici un outil web remarquable pour son lien avec la manipulation de traces au sens du SBT. Samotrace combine la notion de widget avec une implémentation directe en javascript des structures définies du SBT.

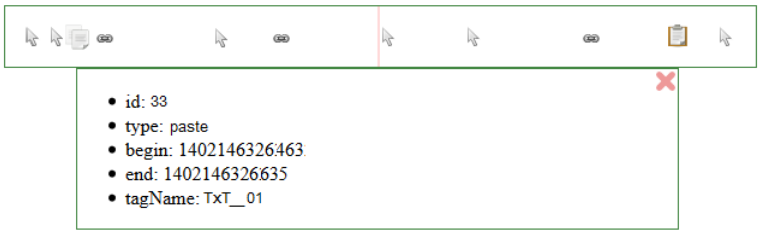

Fig. 3. Exemple de représentation de traces *via* Samotrace [5]

Il s'agit d'un visualisateur graphique implémentant la notion de trace au sens des *SBT*, à comprendre se basant sur la notion d'obsels au sein d'une trace. Bien qu'il ne s'agisse pas directement d'un outil dédié à création d'indicateurs, il fournit un ensemble très solide d'outils permettant une manipulation intuitive et graphique des traces, facilitant grandement la compréhension de ces dernières par un utilisateur ne connaissant pas les obsels ou la notion de trace. Trouvable sur le github de son auteur [2], il implémente un framework facile d'utilisation pour un informaticien qui souhaite présenter ses traces. L'interface est personnalisable et le résultat choisi pour la représentation totalement modifiable selon les besoins. L'interface de présentation de traces, sous forme de frise chronologique (cf. Fig.3), est claire et un bon successeur d'abstract-lite [3], autre outil moins généraliste pour la visualisation de traces d'activité. Samotrace propose également possibilité de charger une trace depuis un kTBS, permettant ainsi de faire abstraction de la forme de stockage RDF de ce dernier.

2. https://github.com/bmathern/samotraces.js/tree/master
3. http://134.214.128.53/abstract/lite/man.html

2.3 Récapitulatif

Si on veut les outils spécifiques à la notion de *SBT*, il est important de réunir ici les points primordiaux auxquels devra répondre le langage que nous souhaitons définir. Dans sa thèse présentant ses conclusions sur les *SBT* [9] et les implémentations possibles, L. S. Settouti présente un ensemble d'éléments à considérer pour l'exploitation des traces et compare différentes solutions, dont celle retenue pour le kTBS, RDF et SPARQL, qui a pour point faible dans son système de requêtes les détections temporelles. Dans l'ensemble des solutions concernant la récupération d'informations sur les bases RDF, la notion de temps est d'ailleurs un point d'information parmi d'autres, sans considérations particulières, ce qu'*a contrario* devra prendre en compte notre langage. De façon récurrente, afin de faciliter l'accès aux données au plus grand nombre, reviennent un certain nombre de méthodes. On peut citer la capitalisation des requêtes, comme dans les travaux précédents concernant la génération des indicateurs [3][4][5][10] ou pour SWIP [13]. Il y a également les modèles qui servent à guider l'utilisateur dans son raisonnement pour la construction d'indicateurs [2] utilisés dans ces mêmes outils.

Du côté du web, on retrouve des approches grandement influencées par le fait qu'on ne connait pas spécialement ce que l'utilisateur va demander et surtout comment. Les approches se couplent alors souvent avec des méthodes de recommandation (approches moteurs de recherche) pour affiner la recherche. SQUALL, lui, cherche à s'approcher au maximum d'une formulation de l'utilisateur, et donne une réponse exacte par analyse de l'entrée. L'approche est pertinente en terme de résultats et soulève des problèmes concernant la formulation des données auxquelles on se retrouve confronté aussi dans le kTBS : comment présenter le contenu des traces à l'utilisateur. La solution proposée est assez simple même si elle suppose une connaissance de l'utilisateur sur le contenu de ses bases, en laissant l'utilisateur "marquer l'information comme dans la base", en utilisant au besoin le préfixe RDF pour en simplifier l'écriture. On peut toutefois se demander si cette implication n'entraine pas les mêmes problèmes qu'a rencontré SWIP lors de la confrontation à une base moins rigoureuse que celle de MusicBrainz lorsque le contenu de la base possède quelques libertés. De plus, si SQUALL obtient de bon résultats, l'utilisateur novice doit savoir quoi inscrire dans le champ texte. SQUALL dans son interface ne propose qu'un lien vers une suite d'exemples fonctionnels, inclinant l'utilisateur à plus s'inspirer de certaines de ces requêtes (comme dans le cadre d'une capitalisation de modèle d'indicateurs) pour les copier et modifier selon les besoins, qu'à inscrire spontanément sa question dans un format utile.

3 Présentation du Langage

Dans cette partie, nous commençons par présenter les bases théoriques du langage et nous expliquerons les choix techniques. Après la présentation d'un extrait représentatif de la grammaire (la version complète est disponible en an-

nexe .1), nous donnerons plusieurs exemples de requêtes et indicateurs dans la forme du langage.

3.1 Forme du Langage

En terme de langage, nous retenons une approche structurée du langage naturel, à l'instar de SQUALL [16]. Bien que l'approche *via* une interface soit celle favorisée par les travaux de l'état de l'art sur les indicateurs, l'interface textuelle possède ses avantages. Lors de l'écriture, l'utilisateur qui connait le langage sera plus apte à réutiliser et à modifier une requête existante, et pourra en plus bénéficier de toutes les fonctionnalités qu'un outil d'édition de texte pourra lui fournir. S'approcher du français dans la formulation des requêtes limite au maximum la quantité de formalisme que l'utilisateur doit assimiler avant de créer ses premières requêtes.

Nous avons choisit de nous baser uniquement sur les contraintes du méta-modèle du kTBS pour la formulation du langage de requête afin d'éviter les biais qui pourraient apparaitre lors d'une utilisation pour une application particulière pour un domaine donné. Le langage est ainsi totalement indépendant du modèle des traces utilisées. Cette généricité permet alors de préparer des interfaces plus spécifiques à chaque domaine d'application, sans gêner la compatibilité des requêtes entre les usages. Cependant, il existe un risque que la requête exprimée dans le langage générique ne soit pas la plus intuitive pour le domaine, dans les cas où le domaine permettrait des raccourcis dans la formulation en français. Il resterait toutefois possible que l'interface implémente ces formulations spécifiques pour ces cas particuliers.

En somme, le langage textuel parsé permet de créer des interfaces adaptées allant de l'interface entièrement graphique (boutons, listes dynamiques) à l'interface textuelle similaire à un champ de recherche d'un moteur internet en passant par des formulaires. Cela permet également à quelqu'un connaissant le langage de se passer entièrement d'interface pour écrire ses requêtes dans un format textuel.

3.2 Des Opérateurs Temporels

Les opérateurs temporels sont des opérateurs de base important pour les *SBT*. Or le SPARQL de base ne contient pas d'opérateurs temporels. Le temps est souvent l'objet dans le système triplet "sujet, prédicat, objet" du RDF. Le prédicat porte le sens du temps décrit en objet et dispose de formats multiples (Unité de temps différentes, dates, en jours ou années). Le SPARQL de base ne contient pas d'opérateurs de base concernant spécifiquement une temporalité quelconque, ni le RDF.

Le temps est représenté en RDF dans le kTBS par un certain nombre de triplets associés aux obsels, les éléments de base du *SBT*. Les obsels sont définis en SBT comme soit étant instantanés, soit ayant une valeur de début et une valeur de fin. L'obsel est alors considéré comme ayant une durée définie par une intervalle. Dans le kTBS, vérifier une date correspond à regarder si l'élément

:hasBegin (le début d'un obsel) ou :hasEnd (la fin d'un obsel) équivaut à certaines conditions. Ces conditions sur le temps peuvent être très basiques. Par exemple, une simple condition sur la date à laquelle s'est produit un élément est facile à exprimer et prise en compte facilement. Dans l'article présentant le *SBT* [9], l'auteur énumère les opérateurs qu'il considère comme importants. Il est d'abord mis en avant que les systèmes informatiques utilisent plus particulièrement les opérateurs concernant les intervalles, à l'opposé des opérateurs sur les événements qui ne durent qu'un instant. Ces obsels sont par exemple des clics de souris ou des évènements marquants, ou encore quelque-chose d'une durée inférieure à l'unité de temps proposée. Pour modéliser la totalité des opérateurs sur intervalles existants, il propose de reprendre les opérateurs dits de Allen [17].

Les opérateurs de Allen sont une manière de modéliser de façon complète l'ensemble des opérateurs possibles entre deux intervalles. Il s'agit d'une énumération nommée des différents cas d'intersection entre deux intervalles. On retrouve ainsi 13 opérateurs (7 opérateurs et leurs opposés, vu que l'opposé de *egals* est lui même) : *before, contains, overlaps, meets, starts, finishes, equals, after, during, overlapped by, met by, started by, finished by, equals.* Prévoir dans le langage les conditions relatives à ces 13 opérateurs permet de s'assurer de pouvoir exprimer la totalité des relations possibles entre deux obsels.

Il faut noter que certains sont moins importants dans le sens où ils sont moins utilisés. L'égalité totale d'un intervalle d'un événement par rapport à un autre est quelque chose d'assez improbable dans une application réelle dont l'unité de temps est inférieure au centième de seconde. Il en va de même pour la correspondance exacte avec une des bornes temporelles d'un évènement réel par rapport à un autre. On peut ainsi donner plus d'importance aux opérateurs que sont : *before, contains, overlaps* et leurs inverses *after, during, overlapped by* dans leur présentation à l'utilisateur final.

Il existe enfin des conditions applicables à ces opérateurs. Outre les multiples sens que peut prendre l'opérateur *before* (avant, mais juste avant, avant sans qu'il n'y ai un autre événement entre, etc...), on peut chercher à connaitre les événements qui se sont déroulés avant une date donnée, ou d'un temps relatif à celui de l'événement considéré.

Par un travail préliminaire à la définition de la grammaire du langage, nous avons défini une traduction littérale des opérateurs de Allen de la façon suivante :

1. before(o1, o2) : l'obsel o1 est situé temporellement avant l'obsel o2. o1 et o2 ne se suivent pas forcément. before(o1, o2) est vrai si la fin de o1 est située après o2.
2. after(o1, o2) : opposé du précédent. L'obsel o1 est situé après l'obsel o2. o1 et o2 ne se suivent pas forcément.
3. equals(o1, o2) : l'obsel o1 et o2 ont le même début et fin.
4. finishes(o1, o2) : la fin de o1 est la fin de o2, et o1 commence pendant o2.
5. finished by(o1, o2) : inverse du précédent. La fin de o1 et o2 sont en même temps, mais o1 commence avant o2.
6. starts(o1, o2) : le début de o1 est en même temps que o2, mais o2 se termine après.

7. started by(o1, o2) : l'inverse du précédent : le début de o1 est en même temps que o2, mais o1 se termine après.
8. meets(o1, o2) : la fin de o1 se déroule en même temps que le début de o2.
9. met by(o1, o2) : la fin de o2 se déroule en même temps que le début de o1.
10. overlaps(o1, o2) : o1 commence avant o2, mais o1 se termine pendant o2.
11. overlaped by(o1, o2) : o1 commence après o2, mais o2 se termine pendant o1.
12. contains(o1, o2) : o2 commence strictement après o1, et o2 se termine strictement avant o1.
13. during(o1, o2) : o1 commence strictement après o2, et o1 se termine strictement avant o2.

Auxquels on rajoute les trois opérateurs suivants :

1. within(o1, o2, temps) : pose une durée maximum de différence entre o1 et o2. La différence est posée comme l'écart entre les différentes bornes de chaque intervalles.
2. successor(o1, o2) : o1 successeur de o2. Le successeur de o1 est défini comme le premier obsel o2 dont le début se déroule en même temps ou après la borne de fin de l'intervalle o1.
3. predecessor(o1, o2) : o1 précède o2. o1 est le prédécesseur de o2 si o1 est le premier obsel se terminant avant le début de o2.

La totalité des 16 opérateurs ainsi définis sont bien sûr totalement combinables entre eux pour obtenir un ensemble quasi-complets pour définir les relations entre deux obsels. Il manque en effet la simple négation des opérateurs (*n'étant pas suivi par*, *strictement suivi par*, ...), mais qui peut se faire simplement en SPARQL 1.1 *via* un filtre.

Nous avons codé ces opérateurs sous forme de matching complet avec leur équivalent SPARQL, et considérés comme utilisables de base pour la grammaire du langage décrite ci-après. Le SPARQL ainsi que ces 16 opérateurs supplémentaires forment le langage pivot permettant de pouvoir facilement, à condition de connaitre le langage SPARQL et le contenu de la trace, requêter les obsels d'un kTBS.

3.3 Langage pour le Langage

Si on souhaite s'inspirer d'une formulation en langage naturel, la question se pose de savoir quelle langue naturelle utiliser pour base de ce langage. Les deux choix à considérer et départager sont le français et l'anglais, principaux candidats du fait du public cible pour le langage. Le français reste le langage d'utilisation directe pour les applications actuelles du kTBS, et des contenus actuels des bases de données kTBS auxquels nous avons accès. Même si nous avons retenu l'utilisation du français pour le langage, l'anglais dispose de nombreux arguments. Il s'agit de la langue principale de publication scientifique, et de nombreux travaux (comme ceux de Montague [14] par exemple) se sont

penchés sur l'interprétation systématique du langage. C'est pourquoi nous expliquerons aussi brièvement ultérieurement les possibilités et problèmes pouvant être rencontrés pour porter notre langage en anglais.

3.4 Grammaire du Langage

La grammaire du langage se présente sous forme de parser packrat, ou parsing expression grammar en anglais (PEG). Il s'agit d'une grammaire qui considère le parsing du texte en son entièreté, *via* un ensemble de règles. La grammaire commence par une première règle, dite de départ, et essaye de trouver la combinaison de règles permettant de parser l'entièreté du texte. Les règles sont sous la forme d'un mélange entre des textes à retrouver dans le parser et d'autres règles imbriquées (ou 'token'). Cela permet de pouvoir réaliser l'ensemble des grammaires que pourraient exprimer un parser LL(k) ou LR(k). Par exemple, examinons la ligne suivante :

départ = " Je veux " suite point

"départ" est le nom de la règle définie dans la ligne. Elle est composée de 3 éléments, qui sont une chaine de caractères, " Je veux ", et de deux règles : suite et point. Chaque règle est définie une fois et une seule, lorsque son nom est suivi par un caractère '='. Si une règle permet plusieurs possibilités de parser l'entrée, chaque possibilité est signalée par un retour à la ligne et un caractère slash. Pour signaler une répétition, ou un élément qui se répète, la chaine de caractères ou le nom de la règle est suivi d'un '+' pour exprimer une répétition d'au moins une fois, d'un '*' pour exprimer une répétition de zéro à plusieurs fois, d'un '?' pour indiquer un élément optionnel. Parfois, la chaine de caractères peut correspondre à une variable (nom d'un obsel particulier, condition sur l'obsel), la variable sera alors indiquée en majuscules.

Extrait de la Grammaire d'une Requête

```
liste_requête  = début* select?

début         = début_select1
               / début_select2
               / début_nommé1
               / début_nommé2
               / début_soit
               / début_parmi

 select          = "Je sélectionne les requêtes : " (NUM, " : ")+ ( NOM, " : " ) "."

début_select1  = " Je cherche à " action nommage point
début_select2  = " Je veux " action nommage point
début_nommé1   = " Je nomme " NOM_RES "les obsels" conditions_obsels* point
```

```
                / " Je nomme " NOM_RES " les attributs " conditions_attributs* point
début_nommé2    = " J'appelle " NOM_RES "les obsels" conditions_obsels* point
                / " J'appelle " NOM_RES "les attributs" conditions_attributs* point
début_soit      = " Soit " NOM_RES action point
                / " Soit " NOM_RES " = " equation point_virgule
début_parmi     = " Parmi " NOM_RES début_select1
                / " Parmi " NOM_RES début_semect2
                / " Parmi " NOM_RES début_nommé1
                / " Parmi " NOM_RES début_nommé2

 action          = " récupérer les obsels " conditions_obsels*
                 / " récupérer les attributs " NOM_ATT? conditions_attributs*
                 / " compter le nombre d'obsels " condition_attributs*
                 / " compter le nombre d'attributs " NOM_ATT? conditions_attributs*
                 / " récupérer les " VALL_ATT conditions_attributs*

conditions_obsels = " de type " NOM_TYPE
                  / " ayant un attribut " NOM_ATT?
                  / " ayant un attribut " NOM_ATT? " de valeur " VALEUR_ATT
                  / " n'ayant pas un attribut " NOM_ATT?
                  / " n'ayant pas un attribut " NOM_ATT? " de valeur " VALEUR_ATT
                  / " commançant après " DATE
                  / "suivi" within? "par un obsel " condobs_temp*
                  / "directement suivi par un obsel " condobs_temp*
                  / " précédé par un obsel " condobs_temp*
                  ... autres opérateurs temporels
                  / ", "
                  / "et "

condobs_temp      = " de type " NOM_TYPE
                  / " ayant un attribut " NOM_ATT?
                  / " ayant un attribut " NOM_ATT? " de valeur " VALEUR_ATT
                  / " n'ayant pas un attribut " NOM_ATT?
                  / " n'ayant pas un attribut " NOM_ATT? " de valeur " VALEUR_ATT
                  / " commançant après " DATE
                  / "lui même suivi" within? "par un obsel " condobs_temp*
                  ... autres opérateurs temporels
                  / ", "
                  / "et "

conditions_attributs = VALL_ATT
                       / "de valeur" VALL_ATT
                       / VALL_ATT "de " NOM

within            = " dans l'heure "
```

```
                    / " dans les " VALEUR " jours qui suivent"
                    / " dans les " VALEUR " secondes qui suivent"
                    / " dans les " VALEUR " heures qui suivent"
                    / " dans les " VALEUR " milisecondes qui suivent  "

point             = " . "
point_virgule     = " ; "

nommage           = " que je nomme " NOM
                  / " nommé " NOM

equation          = CHAINE_SPARQL
```

La formulation de la requête peut être un peu lourde pour certains des opérateurs temporels pour les personnes qui ne sont pas habitués aux opérateurs de Allen. Cependant, les principaux utilisés sont *suivi* et *directement suivi* (et *précédé* et *directement précédé*), ce qui ne devrait pas poser de problème d'utilisation à l'utilisateur non-informaticien par leur définition proche du sens français. Une version plus complète de la grammaire se situe en annexe (cf. annexe .1).

3.5 Exemples et Possibilités

La grammaire en elle-même peut paraitre difficile d'utilisation et certaines "variables" utilisées sont probablement assez obscures sans exemple. La grammaire propose une phrase simple avec un ajout incrémental de conditions sur ce que l'on veut récupérer. On peut vouloir récupérer un ensemble d'obsels (un ensemble peut être composé d'un unique obsel), un ensemble de valeurs ou calculer une valeur particulière. Les phrases peuvent se combiner.

Chaque phrase est une sous-requête de la requête totale. Par défaut, on récupère chaque réponse de chaque sous-requête pour le résultat final. Le résultat est ainsi un ensemble composé d'une part d'un ensemble de traces, et d'autre part d'un ensemble de valeurs calculées. Si l'utilisateur veut des éléments en particulier, il peut utiliser la règle *select*, qui permet de ne prendre que les résultats de requêtes qui l'intéressent en nommant la requête, soit par son nom, soit par son numéro. Le nom est donné par la règle de nommage ou par le *début* permettant de donner un nom : *Soit*, *Je nomme*, *J'appelle*. Le numéro est un numéro attribué par défaut à chaque requête selon son ordre de déclaration, qu'il ait été nommé ou pas.

Nommer une sous-requête à plusieurs avantage. Le nom permet de mieux reconnaitre sa réponse correspondante dans la solution générale, et permet de réutiliser ce résultat de sous-requête dans une autre sous-requête en y faisant référence.

Exemples de Sous-Requêtes Simples

La requête la plus simple que permet le langage est la suivante :

```
"Je veux récupérer les obsels."
```

qui récupère la totalité de la trace. On remarque qu'on cherche à exprimer un ensemble de conditions que les éléments trouvés doivent respecter, et non un élément particulier de la trace. En suivant la grammaire, il s'agit du cas le plus simple, où la répétition du nombre de condition est de 0. Si on veut rajouter une condition, on peut soit donner un type, soit une condition sur les attributs, soit une condition temporelle sur un autre obsel par rapport à celui actuellement considéré.

Pour ajouter une condition de type, il suffit d'accoler la condition "de type" à la phrase. On obtient ainsi :

```
"Je veux récupérer les obsels de type "SimpleObsel"."
```

On peut ajouter autant de conditions de ce type que l'on veut. Généralement, mettre plusieurs fois cette condition mène à un retour vide, du fait qu'un obsel n'a qu'un type. Cependant, un type peut dériver d'un autre type.

Pour ajouter une condition sur les attributs, il suffit d'ajouter à la phrase la condition "ayant un attribut", ou n'ayant pas un attribut. L'attribut peut être suivi par le nom de l'attribut, puis par la valeur recherchée. Les phrases suivantes sont toutes valides :

```
"Je veux récupérer les obsels de type "SimpleObsel" ayant
un attribut de valeur "Henri"."
"Je veux récupérer les obsels de type "SimpleObsel" ayant
un attribut 'id'. "
"Je veux récupérer les obsels de type "SimpleObsel"
ayant un attribut 'chat' de valeur 'irc' . "
```

Pour ajouter une condition temporelle sur un autre obsel, on utilise un des opérateurs de Allen.

```
"Je veux récupérer les obsels suivi par un obsel de type
"SimpleObsel" .
"Je veux récupérer les obsels ayant un attribut "chat",
suivi par un obsel de type "SimpleObsel"."
```

On peut à nouveau appliquer les mêmes conditions au nouvel élément "obsel" déclaré, sous les mêmes contraintes que l'objet principal de la phrase, sauf pour les opérateurs temporels. Pour savoir à quel obsel l'opérateur temporel se réfère, les obsels déclarés après le premier utilisent "lui même" en plus de la phrase précédemment utilisée. Dans les phrases suivantes :

```
"Je veux récupérer les obsels de type A suivi par un obsel de type
B, lui même suivi par un obsel de type C.
"Je veux récupérer les obsels de type A suivi par un obsel de type
B, précédé par un obsel de type "cours".
```

La première fait référence à un obsel de type A suivi par un obsel de type B, B qui est suivi par un obsel de type C. La seconde indique qu'on recherche les obsels ayant au moins un successeur de type B et étant après un obsel de type "cours" en terme de chronologie. Cette spécificité permet de différencier le sujet de l'opérateur temporel. Cela implique que le langage ne permet pas à faire des multiples conditions temporelles sur les obsels servant de conditions temporelles à un autre obsel. Ce cas problématique sera évoqué plus tard dans la partie Discussion. Toutefois, pour la majorité des requêtes, un utilisateur n'aura pas besoin de spécialement plus.

Il peut être intéressant de noter que chaque condition est interchangeable avec une autre. On peut déclarer le type après les conditions sur les attributs. Pour le cas d'une suite d'opérateurs temporels d'obsels, la formulation implique que les conditions s'appliquent au dernier élément "obsel" déclaré. On remarque aussi que les "," et les "et" n'ont aucune sémantique en eux même et ne servent qu'à fluidifier la formulation des sous-requêtes.

Combiner les Requêtes. Pour pouvoir combiner les requêtes, il est préférable de leur donner un nom. Pour cela, les requêtes *Je nomme*, *Soit* permettent de définir un nom de référence. Les autres *début* doivent utiliser le *que je nomme* à la fin de la requête. Une fois les éléments nommés, on peut les réutiliser dans une requête *Soit* pour faire un calcul, ou les utiliser dans une requête *Parmi*, pour rajouter des conditions.

Pour les calculs via Soit, il faut avoir récupérer un certain nombre de variables des autres requêtes. La requête suivante :

```
"Je veux récupérer les obsels de type note que je nomme Notes."
"Soit A les attributs "resultats" de Notes."
"Soit Moyenne = AVG( A )."
```

permet de calculer la moyenne des valeurs d'attributs résultats des obsels de type "note". (soit la moyenne des notes). Le calcul pour l'équation est exprimé dans la même manière que dans SPARQL.

En utilisant Parmi, on peut rajouter des conditions supplémentaires à un ensemble d'obsels vu précédemment. La requête suivante :

```
"Je veux récupérer les obsels de type note que je nomme Notes."
"Parmi Notes, je veux les obsels ayant un attribut de valeur
"Eric", que je nomme Notes_Eric.
```

récupérera dans Notes_Eric les obsels ayant un type "note" et ayant un attribut de valeur "Eric". Ce qui selon la base, pourrait correspondre à l'ensemble des notes du dit Eric.

Equivalences de Requêtes. On peut remarquer que la grammaire propose un certain nombre de *début* qui sont totalement équivalent dans leur utilisation. L'utilisateur pourra utiliser indifféremment les débuts proposés. Ces éléments, des synonymes, permettent dans le cas d'un système de complétion textuelle, par exemple, de permettre à l'utilisateur de trouver plus rapidement un début valable pour sa requête.

3.6 Sauver et Garder : Capitalisation des Connaissances

Sauvegarder la requête peut être fait par un simple enregistrement textuel. Cependant, pour plus de facilité à la réutilisation, il parait préférable de sauvegarder sous un format similaire à celui utilisé pour les indicateurs. En fournissant des informations supplémentaires que sont un "nom" et un "descriptif textuel", on peut construire un index des requêtes et ainsi les réutiliser plus rapidement. Il faudrait cependant noter une information supplémentaire. Soit le modèle de la trace sur laquelle la requête doit se lancer, soit un lien vers un exemple de trace. Une format de sauvegarde conseillé est ainsi un quadruplet Q tel que

$$Q = \{n, des, req, m\} \tag{1}$$

où n est le nom donné à la requête, *des* une description textuelle de la requête, *req* la formulation dans le langage et m le modèle de trace utilisé.

4 Implémentation

Pour l'implémentation, nous avons étudié plusieurs langages pour choisir le plus pertinent pour une première implémentation. Parmi ces langages, on cite python, javascript et java ou C. Pour interroger une base RDF, nous avons trouvé plus pertinent de privilégier le python ou le javascript. Ces deux langages permettent une implémentation légère plus facile au niveau des navigateurs. De plus, un certain nombre d'outils liés au kTBS utilisent ces technologies (Samotrace, par exemple).

Voulant appliquer les technologies aux traces stockées sous format RDF sur un kTBS en ligne, dans le cadre de traces d'un MOOC sur une plateforme en ligne, le choix s'est dirigé sur une première implémentation en javascript pour le parser, et une implémentation en html+javascript pour les interfaces. L'utilisation du html et javascript devant faciliter l'adaptation directe pour la plateforme du MOOC.

PEG.js. Parser Générator for Javascript [18] est un générateur de parser qui utilise des principes très similaires aux outils unix flex et bison, mais dans le but de générer un parser en code javascript. Il prend en entrée quelque chose décrivant une grammaire PEG, avec pour chaque entrée une possibilité d'utiliser du code javascript pour préciser ce que le programme remonte pour générer le code SPARQL exécutant les parties de code écrites par l'utilisateur. L'outil,

généraliste et disponible en ligne et en package unix, s'est révélé pratique et facile d'utilisation. Son interface en ligne, notamment, est très pratique et permet de choisir les principaux paramètres qui permettent de personnaliser le parser.

Cependant, malgré la présence du générateur, la transformation du langage écrit dans la première partie en réel parser à posé plusieurs problème techniques. Du fait de certains mécanismes proposés précédemment, dont le nommage, le simple parsing textuel ne permet pas de passer en pseudo langage naturel au langage SPARQL, et nous avons eu besoin de mettre en place une structure supplémentaire pour les éléments les plus avancés du langage. Dans cette partie, on présentera ainsi les différentes interfaces et structures de données implémentées et utilisées pour le parser.

A l'heure actuelle, seule une partie du langage décrit en partie 3 a été complètement implémenté. il manque une grande partie des opérateurs temporels dans le langage en pseudo-français. Les conditions de bases sur le temps (suivit, précédés) sont cependant fonctionnels.

4.1 Opérateurs Temporels

Nous avons commencé par l'implémentation de l'ensemble des 16 opérateurs décris, l'ensemble des opérateurs de Allen et les 3 ajouts que sont *within*, *successeur* et *predecessor*. Nous avons utilisé le parser peg.js, bien que l'ensemble des opérateurs auraient pu se résumer à un ensemble de chercher/remplacer dans le texte du formalisme proposé. Nous avons ensuite testé et validé par une base de traces-jouet contenant l'ensemble des cas possibles d'intersection (et de non intersection) d'intervalles.

4.2 Sous-Requêtes Simples

L'implémentation de sous-requêtes simples n'a pas posé beaucoup de problèmes concernant le codage du parser. L'essentiel des difficultés rencontrées étaient dûes au niveau de permissivité dans ce qui est accepté comme valable dans les appariements *via* expressions régulières. Nous avons choisi par exemple de considérer indifféremment les requêtes en minuscules et celles en majuscules. On peut aussi utiliser un nombre illimité d'espaces entre les mots du langage. Au niveau des variables, un certain nombre de choix ont dû être faits :

1. Les noms de sous-requêtes sont des chaines de caractères sans caractères spéciaux et sans espaces. On peut choisir de manière sûre les caractères alphanumériques non accentués. Les accents sont tolérés, mais cela reste source d'erreurs lors des sauvegardes et chargements lors du passage d'un environnement à un autre. Un nom de variable ne commence pas par un chiffre.
2. Les valeurs des variables pour l'ensemble des conditions peut être soit une chaine de caractères sans espace, soit un format d'élément de SPARQL, ceci pour s'assurer de pouvoir tout réutiliser d'une base donnée à une autre. Si on veut faire une condition sur une chaine de caractères, il faut l'entourer

de simple guillemets. Les guillemets sont nécessaires au parser pour qu'il puisse délimiter où commence et où s'arrête la chaine de caractères servant d'argument. Sans cette précaution, le parser pourrait considérer la totalité de la fin de la requête comme étant l'argument de la condition. Il faut en effet ne pas exclure la possibilité de rechercher dans les traces des mots du langage.

3. Le parser dispose de deux variables globales. La première est ce qu'on appelle la base par défaut, qui est un lien vers la base à interroger. Le second sert à exprimer le préfixe par défaut rajouté avant chaque valeur dans les conditions. Le préfixe est généralement une URI vers le modèle de la trace.

4.3 Composition de Requêtes

La composition de requêtes amène la problématique de la gestion du nom au travers du parser, afin de pouvoir utiliser les entrées "Parmi" ou "Soit". Pour ce faire, nous définissons une structure de base représentant la sous-requête, la requête ayant alors une liste de ces sous-requêtes. La sous-requête est définie comme un quadruplet SR.

$$SR = \{id, name, SELECTBLOCK, dependances\} \tag{2}$$

où *id* est le numéro attribué automatiquement selon sa position dans la requête, *name* le nom donné si présent dans la sous-requête. Si la sous-requête n'a pas de nom, on considère que ce dernier est vide, et on vérifie uniquement les références *via* l'*id*. *SELECTBLOCK* est le code SPARQL temporaire déduit par le parser. *dépendance* est une liste de couples dont le premier élément est une chaine de caractère et dont le deuxième élément est un entier indiquant le type de remplacement à effectuer.

Chaque sous-requête parsée renvoie une structure remplie à la requête. La requête doit ensuite résoudre chaque dépendance. Pour ce faire, elle remplace la référence par le code correspondant dans le *SELECTBLOCK*.

Cela implique l'existence de marqueurs dans le *SELECTBLOCK* de chaque structure *SR*. Ces marqueurs sont générés automatiquement sur la base du nom et du type de remplacement. Si on fait une condition sur une valeur d'attribut, qui est généralement l'objet d'un triplet RDF d'un obsel donné, la référence est : id_ref+ID+_o si la sous-requête référée n'a pas de nom, id_ref+NOM+_o sinon.

La requête dispose à la fin du parsing d'un ensemble E composés de structures SR. On applique sur E un algorithme (cf. annexe .2) pour résoudre la totalité des dépendances. A la fin de l'algorithme, il suffit de fusionner l'ensemble des *SELECTBLOCK* de chaque sous-requête que l'on veut garder pour le résultat. Cette manière de faire interdit cependant les références croisées entre requêtes. L'implémentation de l'algorithme gère la détection de l'auto-référence et de référence invalide (vers un élément n'existant pas).

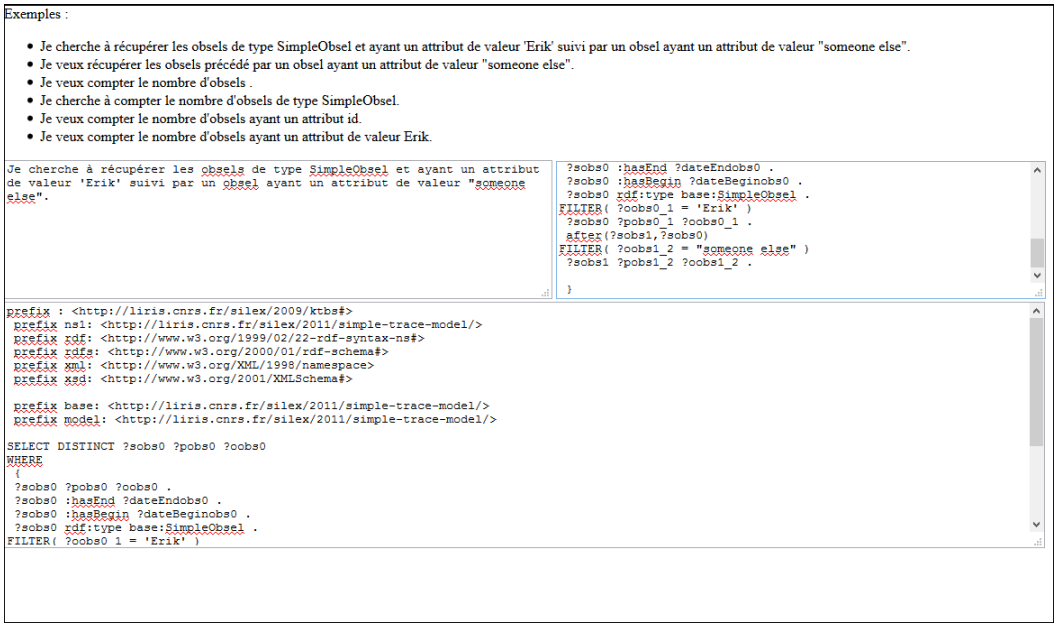

Fig. 4. Interface textuelle du parser

4.4 Interface Textuelle

La première interface faite a été l'interface textuelle (cf. Fig.4). Destinée aux utilisateurs connaissant le langage, elle renvoie directement le résultat SPARQL utilisable par un quelconque SPARQL end-point supportant le SPARQL 1.1. L'interface propose trois champs. Celui au milieu à gauche est celui acceptant le langage présenté en abrégé ici (en version complète dans l'annexe .1). Le champ situé à sa droite est le résultat en langage pivot (SPARQL augmenté des opérateurs temporels) obtenu par le parser. Le troisième champ texte contient la requête SPARQL compatible avec SPARQL 1.1.

L'utilisateur peut modifier les trois champs textes. Les modifications du premier champ sont répercutés sur les deux suivants, celles du second champ sur le résultat SPARQL, ce qui permet de rajouter rapidement des relations temporelles supplémentaires dans les conditions au besoin. Afin d'aider l'utilisateur dans l'écriture de ses premières requêtes, on lui met à disposition un ensemble d'exemples de requêtes fonctionnelles (cf. haut de Fig.4).

4.5 Interface Graphique

L'interface graphique (cf. Fig.5) dispose de mécanismes permettant d'envoyer les requêtes sur une base définie *via* l'interface. En haut, dans le cadre gris, se situe la liste de sous-requêtes que l'on veut envoyer. Par défaut, seule une requête, entourée de noir, est envoyée à la base kTBS. Cette requête est appelée le focus.

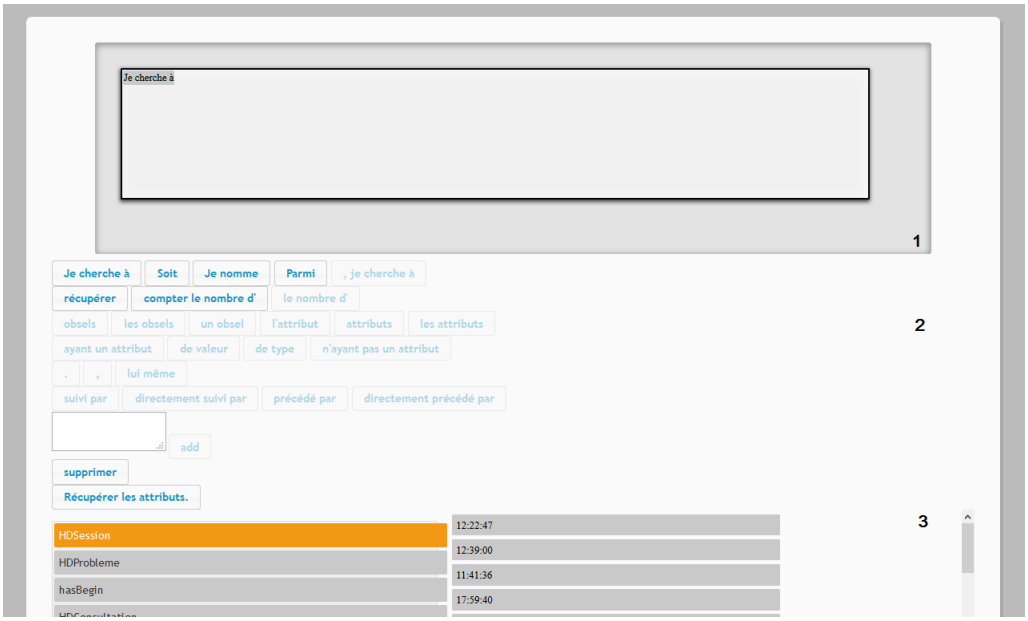

Fig. 5. Interface graphique du parser

Sous le cadre contenant les requêtes (cf. 1 sur Fig.5), sont disposés un ensemble de boutons (2). Ces boutons sont des mots du langage. Les boutons non-grisés sont les boutons permettant d'avancer dans la construction de la requête sans enfreindre une règle du parser.

Enfin, en dessous, une double liste (3) permet d'explorer le contenu des traces. La première colonne permet de naviguer dans les attributs disponible. Lorsque des éléments de cette colonne sont sélectionnés, les valeurs possibles pour ces attributs apparaissent dans la seconde liste. Ces éléments peuvent ensuite être utilisés dans l'interface pour les intégrer à la sous-requête. Cette double liste est provisoire et les informations qu'elle contient sont prévues d'être représentés graphiquement grâce à Samotrace.

5 Discussion et Perspectives

Ce stage au sein de l'équipe SILEX dans le cadre du projet COAT s'articule autour de l'accessibilité de l'interrogation du contenu des traces. De ce fait, nous proposons un langage destiné à abstraire l'usage du SPARQL à l'utilisateur, en s'inspirant des méthodes utilisées dans les autres travaux du domaine du web cherchant à rendre accessible les bases RDF au plus possible [15] [12]. Dans un premier point nous reviendrons brièvement sur les résultats et les interfaces produites, avant d'enchainer sur certains points du langage et choix effectués.

5.1 Contributions

Le langage dans sa définition est un sous-ensemble très restreint de formulations possibles se basant sur le langage français. À l'aide d'un certain nombre d'amorces de phrase, on guide l'utilisateur sur la forme globale de sa formulation de requête, en s'aidant du français pour faciliter l'apprentissage.

Le parser, actuellement incomplet, de ce langage, prend la forme d'une fonction javascript. Elle prend une entrée textuelle et génère une requête SPARQL qui peut ensuite être envoyée sur le kTBS.

Cependant, une simple interface textuelle n'est généralement pas suffisante pour celui qui ne connait absolument pas le langage. Face à un champ textuel vide et sans aucune autre indication, il est peu probable que l'utilisateur arrive à utiliser l'outil efficacement. Pour remédier à cela, nous pouvons choisir de faire comme SQUALL, en donnant un certain nombre d'exemples, ou de faire une interface qui aura pour but d'encadrer la saisie de l'utilisateur.

L'encadrement de la saisie de l'utilisateur peut se faire de plusieurs manières. Un moyen simple de guider l'utilisateur est la simple auto-complétion d'éléments. Le parseur généré par peg.js [18] se prête d'ailleurs bien à l'exercice, proposant un retour d'erreur complet sur ce que le parseur attend à la suite de l'élément donné. Un autre moyen est de faire une interface complètement graphique, et disposant des fonctionnalités principales du langage (cf. Fig.5). Ce moyen permet de s'assurer que l'utilisateur suit la grammaire et limite les fautes de frappe, mais ralentit considérablement la construction de la requête.

Remarques sur le Langage Il existe plusieurs reproches qui peuvent venir immédiatement à l'esprit en observant la formulation du langage. Le premier reproche est une lourdeur dans la formulation de ce dernier. "Je cherche à récupérer les obsels de type A [...]" fait alambiqué pour un raccourci qui pourrait être simplement "Je cherche les obsels de type A [...]", voir encore "Je cherche les A [...]" . Si la formulation "Je cherche les obsels de type A" peut paraître lourde, la reformulation raccourcie "Je cherche les A" pose le problème d'ambiguïté avec les attributs, que l'on pourrait tout aussi bien récupérer ainsi. Cependant, le "Je cherche les A" au lieu de "Je cherche les obsels de type A" a aussi l'avantage de supprimer momentanément la notion d'obsel ou d'ensemble d'attributs à l'utilisateur final.

De même, lorsque les obsels de la requête ont un grand nombre de conditions, on peut reprocher au langage de perdre sa conformité vis-à-vis de la grammaire française. Cependant, les requêtes basiques (avec un nombre restreint de conditions) devraient pouvoir être utilisés par un utilisateur non-informaticien.

Il a déjà été souligné que le terme obsel et la notion d'attribut puisse être source de rejet immédiat de la part de certains utilisateurs, en tant que termes spécifiques à l'utilisation de l'outil.

Contenu de la base. La présentation des données à l'utilisateur final est un problème que l'on rencontre souvent. À moins d'être le créateur des données

ou d'avoir travaillé longtemps dessus, l'utilisateur final ne connait généralement pas le contenu des traces, ni le modèle avec précision. Dans squall2sparql [16], l'auteur du langage choisit de laisser à l'auteur de la requête le soin de "bien remplir" ces éléments qui sont en fait spécifiques au modèle. L'utilisateur doit faire un léger effort de recherche pour retrouver le nom et la façon d'écrire le contenu pour utiliser SQUALL. Le même problème se pose ici pour récupérer le contenu des traces.

La solution actuelle, proposer un préfixe par défaut que le parser utilisera en y accolant le nom de l'attribut ou la valeur de la variable n'est satisfaisante que si l'utilisateur est maître du contenu de sa base de traces et des noms de chaque élément dans ses traces.

Il est important d'afficher à l'utilisateur ce qu'il peut récupérer dans ses bases [4]. Plusieurs propositions sont possibles pour faciliter la récupération de ces valeurs ou attributs. La première, qui est d'afficher le contenu des traces par attribut et par valeur, n'aide que les personnes qui ont une idée de ce qu'elles cherchent. Si la personne ne se souvient que très mal du contenu de ses traces, ou ne les connait pas, l'aspect graphique parait le plus pertinent. Nous avons identifié deux possibilités de visualisation, dont l'implémentation serait liée à Samotrace. Il serait possible de représenter le modèle sous forme de trace graphique consultable dynamiquement à l'aide de la souris. L'autre méthode, où l'on affiche une 'trace-jouet' qui contiendrait des exemples d'instanciation de la trace, avec au moins un obsel de chaque type existant, aurait l'avantage de ne pas montrer que le modèle, mais aussi des exemples de ce qu'on peut trouver dedans.

Enfin, une autre solution serait, à l'aide d'un travail d'ingénierie supplémentaire sur les traces, de générer un ensemble de relations entre des éléments affichés à l'utilisateur (en français) et des éléments de la base (écrit en RDF), ce qui ressemble à la méthode utilisée dans EM-AGIIR [5], où l'on demande à l'utilisateur de nommer ses besoins, puis ensuite de rechercher les éléments dans la base de traces.

Sur les Interfaces. L'interface mot à mot s'avère parfois laborieuse à utiliser. De plus, la recherche d'éléments peut également l'être. Il existe un moyen potentiel pour encore faciliter les premières utilisations du langage. Au lieu de proposer mot par mot, on pourrait proposer d'ajouter une liste de conditions : des morceaux de phrases. On se situerait à mi-chemin du pattern complet et du langage mot à mot, en proposant à l'utilisateur de créer son pattern particulier à l'aide de briques de conditions. (cf. annexe .3). Il obtiendrait visuellement un arbre, qui lui permettrait de mieux voir quelles conditions affecteraient quel élément de la phrase. Il est à noter que cela ne réduirait pas le problème posé par la sélection du contenu de la base discuté plus haut.

Portage du Français vers l'Anglais. Le langage fonctionne sur une base de français, qui s'éloigne un peu sur certains points de la grammaire anglaise. La mise en place de synonyme comme en français et le travail sur le parser reste une charge certaine de travail.

La plupart des mots clef du langage peuvent être directement traduits; "Je veux" par "I want to", "récupérer" par "get" ou "compter" par "count". Les phrases simples peuvent alors être traduites mots à mots et rester compréhensibles, comme "Je veux récupérer les obsels de type 'chat' ayant un attribut subject de valeur "Henri"." pourrait être traduite "I want to get all the obsel of type 'chat' with an attribute subject of value "Henri"". Dans le cas des phrases composées, par contre "I want to get all obsel of type "enter" that are followed by an obsel of type "message" , followed by an obsel of type "quit"", l'obsel de type "quit" fait logiquement référence à l'obsel de type "message", là où le français laisse une ambiguïté coupée par une règle imposée par le langage théorique.

5.2 Évaluation et Perspectives

Durant le mois et demi de stage restant, nous commencerons par nous pencher sur l'évaluation du langage que nous proposons pour le valider. Ce langage soulève également plusieurs perspectives, dont la première, que nous traiterons également dans ce stage, est la création de pattern d'indicateurs.

Évaluation

Afin de valider les propositions, il faut vérifier le langage sur plusieurs points. Il faut tout d'abord vérifier que le langage permet de définir tout les indicateurs que l'on souhaite. Il faut aussi vérifier que le langage soit utilisable par un utilisateur non-informaticien.

Pour évaluer la complétude du langage, il y a deux possibilités. La première est expérimentale et consiste à énoncer la formulation dans notre langage des indicateurs exprimés par des potentiels utilisateurs, ce qui correspond aux premiers tests que nous avons entrepris. Bien que ne prouvant pas l'expressivité totale du langage dans le cadre de la création d'indicateurs, cela permet de prouver que le langage peut répondre à un certain nombre de besoins. Par ailleurs, dans le même ordre d'idée, il serait intéressant de voir si le langage permet de définir l'ensemble des indicateurs ou type d'indicateurs répertoriés par l'état de l'art du groupe Kaléidoscope [1]. La seconde méthode est de passer par une preuve théorique, en évaluant par exemple l'expressivité du langage par rapport à un autre connu comme pouvant répondre aux besoins d'expressivités (par exemple, le SPARQL).

Pour évaluer l'utilisabilité du langage, il convient d'en évaluer la facilité d'utilisation par les utilisateurs cibles, les non-informaticiens. Nous avons discuté avec l'auteur non-informaticien expert en anatomie du MOOC FOVEA [4], Patrice Thiriet, qui s'est montré très intéressé bien qu'ayant montré des réserves sur la manière actuelle de visualiser le contenu des traces (sous forme de listes, où la recherche d'éléments peut être fastidieuse) et sur l'utilisation de termes

4. http://anatomie3d.univ-lyon1.fr/webapp/website/website.html?id=2372715

spécifiques comme la notion d'attributs, d'obsels ou de traces, évoquant que cela puisse être rébarbatif pour les utilisateurs les moins persistants. Une réunion de travail avec lui est prévue en juillet afin d'évaluer l'implémentation du langage.

Perspectives

Pattern de Requête. Les patterns de requêtes sont un bon moyen de permettre la réutilisation de requête. Ces patterns peuvent ensuite être stockés dans un catalogue les indexant selon leur usage (la classification de Kaléidoscope peut être utilisée [1]) pour améliorer l'utilisabilité du langage. La généralisation d'un indicateur pour sa réutilisation, soit dans une trace de même modèle (sans adaptation), soit dans une trace présentant des similarités, demande un travail supplémentaire de formalisation à partir d'indicateurs déjà formalisés dans notre langage pour en dégager les parties qui sont à changer selon les traces et les conditions que doit valider l'élément remplaçant.

Perspectives à long terme. La conception de ce langage se situe en interface avec le kTBS pour en récupérer des informations. Les requêtes exprimées partent du postulat que l'utilisateur à une idée de ce qu'il veut récupérer dans les traces, ou du moins possède une première formulation de ce qu'il veut pouvoir récupérer. Le langage lui permet alors d'exprimer sa requête en tentant de rester le plus proche possible de cette première formulation. Dans le cadre d'utilisation du kTBS, le langage et son parser pourraient potentiellement être utilisés à plusieurs niveaux : calculer sur des traces les indicateurs pour un profil d'apprenant (son premier objectif), mais aussi permettre à des utilisateurs plus avancés d'utiliser le langage pivot ou ensuite pourquoi pas SPARQL pour enrichir une requête de base générée par le parser et répondre à un besoin plus spécifique. Par exemple, un utilisateur souhaitant faire une transformation pourrait tout d'abord récupérer les données l'intéressant dans une requête générée, puis pourrait alors s'en aider pour compléter sa requête CONSTRUCT servant à faire sa transformation. L'existence d'un parser passant d'un langage formalisé à du SPARQL peut également permettre d'envisager le passage dans l'autre sens, sous condition de respecter des règles de formulation dans la requête SPARQL.

References

1. Dimitrakopoulou, A. (2004). State of the art on Interaction and Collaboration Analysis (D26. 1.1). EU Sixth Framework programme priority 2, Information society technology, Network of Excellence Kaleidoscope,(contract NoE IST-507838), project ICALTS: Interaction and Collaboration Analysis
2. Dimitrakopoulou, A., Petrou, A., Martinez, A., Marcos J. A., Kollias, V., Jermann, P., Harrer, A., Dimitriadis, Y., Bollen, L. (2007). State of the art of interaction analysis for Metacognitive Support and Diagnosis.
3. Gendron, É. (2010). Cadre conceptuel pour l'élaboration d'indicateurs de collaboration à partir des traces d'activité (Thèse, Université Claude Bernard-Lyon I).

4. Djouad, T. (2011). Ingénierie des indicateurs d'activités à partir de traces modélisées pour un Environnement Informatique d'Apprentissage Humain (Thèse, Université Claude Bernard-Lyon I).
5. Diagne, F. (2009). Instrumentation de la supervision de l'apprentissage par la réutilisation d'indicateurs: Modèles et Architecture (Thèse, Université Joseph Fourier - Grenoble).
6. Pierre-Antoine Champin, Yannick Prié, and Alain Mille, (2003). Musette : Modelling uses and tasks for tracing experience. In Béatrice Fuchs and Alain Mille, editors, workshop From structured cases to unstructured problem solving episodes - WS 5 of ICCBR'03, pages 279–286, Trondheim (NO).
7. Alain Mille (2006). From case-based reasoning to traces-based reasoning. Annual Reviews in Control 30(2): 223-232
8. Settouti, L. S., Prié, Y., Champin, P.-A., Marty, J.-C., and Mille, A. (2009). "A Trace-Based Systems Framework : Models, Languages and Semantics".RR report. http://hal.inria.fr/inria-00363260/en/.
9. Settouti, L. S. (2011), Systèmes à Base de traces modélisées - Modèles et langages pour l'exploitation des traces d'Interactions. (Thèse, Université Claude Bernard-Lyon I).
10. Choquet, C., et Iksal, S. (2007). Modélisation et construction de traces d'utilisation d'une activité d'apprentissage: une approche langage pour la réingénierie d'un EIAH. Revue des Sciences et Technologies de l'Information et de la Communication pour l'Education et la Formation (STICEF), 14.
11. Choquet, C., et Iksal, S. (2010). Un Système à Base de Traces pour la modélisation et l'élaboration d'indicateurs d'activités éducatives individuelles et collectives. Mise à l'épreuve sur Moodle. TSI, vol. 29, pages 721–741.
12. Camille Pradel, Ollivier Haemmerle, et Nathalie Hernandez (2011). A Semantic Web Interface Using Patterns: The SWIP System. IJCAI-GKR Workshop, Barcelona, Spain, pages 172–187.
13. swip.univ-tlse2.fr/SwipWebClient
14. Montague, R. (1973). The proper treatment of quantification in ordinary English. In Approaches to natural language (pp. 221-242). Springer Netherlands.
15. Ferré, S. (2012). SQUALL: A controlled natural language for querying and updating RDF graphs. In Controlled Natural Language (pp. 11-25). Springer Berlin Heidelberg.
16. Ferré, S. (2013). squall2sparql: A translator from controlled English to full SPARQL 1.1. Proceedings of the Question Answering over Linked Data lab (QALD-3) at CLEF.
17. Allen, J, F.(1983). Maintaining knowledge about temporal intervals. Commun. ACM, 26(11) (pp 832–843)
18. Site du générateur de parser de type PEG (parsing expression grammar), http://pegjs.majda.cz/
19. Question answering over linked data, http://greententacle.techfak.uni-bielefeld.de/ cunger/qald/
20. Djouad, T., Settouti, L. S., Prié, Y., Reffay, C., et Mille, A. (2010). Un Système à Base de Traces pour la modélisation et l'élaboration d'indicateurs d'activités éducatives individuelles et collectives. Mise à l'épreuve sur Moodle. Revue Techniques et Sciences informatiques, 29(6), 721-741.

6 Annexes

.1 Grammaire complète du Langage

```
liste_requête  = début* select?

début          = début_select1
               / début_select2
               / début_nommé1
               / début_nommé2
               / début_soit
               / début_parmi

 select          = "Je sélectionne les requêtes : " (NUM, " : ")+ ( NOM, " : " ) "."
                / "Je garde uniquement " (NUM, " : ")+ ( NOM, " : " ) "* "."

début_select1  = " Je cherche à " action nommage point
début_select2  = " Je veux " action nommage point
début_nommé1   = " Je nomme " NOM_RES "les obsels" conditions_obsels* point
               / " Je nomme " NOM_RES " les attributs " conditions_attributs* point
début_nommé2   = " J'appelle " NOM_RES "les obsels" conditions_obsels* point
               / " J'appelle " NOM_RES "les attributs" conditions_attributs* point
début_soit     = " Soit " NOM_RES action point
               / " Soit " NOM_RES " = " equation point_virgule
début_parmi    = " Parmi " NOM_RES début_select1
               / " Parmi " NOM_RES début_semect2
               / " Parmi " NOM_RES début_nommé1
               / " Parmi " NOM_RES début_nommé2

 action         = " récupérer les obsels " conditions_obsels*
                / " récupérer les attributs " NOM_ATT? conditions_attributs*
                / " compter le nombre d'obsels " condition_attributs*
                / " compter le nombre d'attributs " NOM_ATT? conditions_attributs*
                / " récupérer les " VALL_ATT conditions_attributs*

conditions_obsels = " de type " NOM_TYPE
                  / " ayant un attribut " NOM_ATT?
                  / " ayant un attribut " NOM_ATT? " de valeur " VALEUR_ATT
                  / " n'ayant pas un attribut " NOM_ATT?
                  / " n'ayant pas un attribut " NOM_ATT? " de valeur " VALEUR_ATT
                  / " commançant après " DATE
                  / "suivi" within? "par un obsel " condobs_temp*
                  / "directement suivi par un obsel " condobs_temp*
                  / " précédé par un obsel " condobs_temp*
                  / " directement précédé par un obsel " condobs_temp
                  / " commencé par un obsel " condobs_temp*
```

```
                    / " commençant un obsel " condobs_temp*
                    / " finissant un obsel " condobs_temp*
                    / " fini par un obsel " cond_obs_temp*
                    / " exactement en même temps qu'un obsel " condobs_temp* condobs*
                    / " superpose un obsel " condobs_temp*
                    / " est superposé par un obsel " condobs_temp*
                    / " durant un obsel " condobs_temp*
                    / " alors que se déroule un obsel " condobs_temp
                    / ", "
                    / "et "

condobs_temp        = " de type " NOM_TYPE
                    / " ayant un attribut " NOM_ATT?
                    / " ayant un attribut contenant la valeur " CHAINE_SPARQL
                    / "n'ayant pas un attribut contenant la valeur "
                    CHAINE_SPARQL
                    / " ayant un attribut " NOM_ATT? " de valeur " VALEUR_ATT
                    / " ayant un attribut " NOM_ATT? " de valeur superieure à "
                    VALEUR_ATT
                    / " ayant un attribut " NOM_ATT? " de valeur inferieure à "
                     VALEUR_ATT
                    / " n'ayant pas un attribut " NOM_ATT?
                    / " n'ayant pas un attribut " NOM_ATT? " de valeur " VALEUR_ATT
                    / " n'ayant pas un attribut " NOM_ATT? " de valeur superieure à "
                    VALEUR_ATT
                     / " n'ayant pas un attribut " NOM_ATT? " de valeur inférieure à"
                     VALEUR_ATT

                    / " commançant après " DATE
                    / "lui même suivi" within? "par un obsel " condobs_temp*
                    / "lui même directement suivi" within? " par un obsel " condobs_temp*
                    / "lui même précédé" within? " par un obsel " condobs_temp*
                    / "lui même directement précédé" within?
                      " par un obsel " condobs_temp
                    / "lui même commencé par un obsel " condobs_temp*
                    / "lui même commençant un obsel " condobs_temp*
                    / "lui même finissant un obsel " condobs_temp*
                    / "lui même fini par un obsel " cond_obs_temp*
                    / "lui même exactement en même temps qu'un obsel " condobs_temp*
                    condobs*
                    / "lui même superpose un obsel " condobs_temp*
                    / "lui même superposé par un obsel " condobs_temp*
                    / "lui même se déroulant durant un obsel " condobs_temp*
                    / " lui même pendant un obsel " condobs_temp
                    / ", "
```

```
                    / "et "

   conditions_attributs = VALL_ATT
                        / "de valeur" VALL_ATT
                        / VALL_ATT "de " NOM

   within            = " dans l'heure "
                     / " dans les " VALEUR " jours qui suivent"
                     / " dans les " VALEUR " secondes qui suivent"
                     / " dans les " VALEUR " heures qui suivent"
                     / " dans les " VALEUR " milisecondes qui suivent  "

point              = " . "
point_virgule      = " ; "

nommage            = " que je nomme " NOM
                   / " nommé " NOM

equation           = CHAINE_SPARQL
CHAINE_SPARQL = "  [^"]* "
/' [^']* '
VALEUR_ATT = ( valeurs possibles variable sparql)
NOM = [a-zA-z]*[^ \n\t]*
```

.2 Algorithme

Cet algorithme présente le processus de traitement des dépendances de la liste des structures issues du traitement par le parser de l'ensemble des sous requêtes. L'implémentation de cet algorithme doit prévoir le traitement des cas où les sous-requêtes font des références croisées ou des références vers des noms non-existants.

Algorithm 1 Algorithme de Remplacement de Références dans les Sous-requêtes

Données : Ensemble E de $SR = (id, name, SELECTBLOCK, dependances)$
Résultat : Requête Sparql valide.

Pour n $\leftarrow 1, , |E|$ **Faire**
  Pour i $\leftarrow 1, , |E[i].dependances|$ **Faire**
    On recherche le nom ou l'id correspondant à E.dependances[i][0] dans l'ensemble E. On nomme cet élément E[k]
    Si $f(0 < |E[k].dependances|$ **Alors**
      On applique le préent algorithme sur E[k]
    Fin si
    Pour chaque élément généré de référence possible id_ref, on remplace par le bon élément dans le SELECTBLOCK.
    On rajoute le SELECTBLOCK de E[k] à la fin de celui de E[i]
  Fin pour
Fin pour

.3 Construction alternative

La présentation en arbre permet de mieux expliciter la hiérarchie et quelle condition s'applique à quel élément.

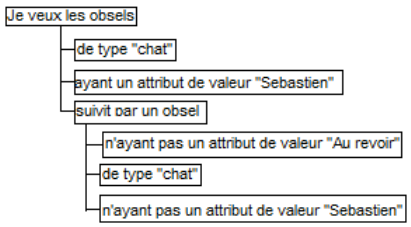

Fig. 6. Une structure hierarchique pour exprimer les conditions sur les obsels.