Projet ROSES
Programme MDCO – Edition 2007

Livrable no D5.1
View language for declarative specification of RSS-XML services

Identification

Acronyme du projet	ROSES
Numéro d'identification de l'acte attributif	ANR-07-MDCO-011-01
Coordonnateur	Paris 6
Rédacteur (nom, téléphone, email)	Dan Vodislav et al. 0134252814 dan.vodislav@u-cergy.fr
No. et titre	D5.1 : View language for declarative specification of RSS-XML services
Version	1
Date de livraison prévue	t0+18
Date de livraison	

Résumé

Ce livrable présente le langage déclaratif de spécification de nouvelles publications (flux) dans ROSES. Une nouvelle publication est définie comme une *vue de composition* (flux virtuel) sur d'autres publications ou comme une matérialisation d'un flux virtuel.

Sont décrites les choix de modélisation dans la définition du langage, en complément du modèle de données et de l'algèbre ROSES, ainsi que la syntaxe complète du langage de publication. Les principales caractéristiques sont illustrées par des exemples d'utilisation.

Table des matières

Introduction

Les services RSS-XML que le système ROSES permet de définir correspondent à la notion de ***publication***. Nous rappelons qu'une publication est un nouveau flux créé dans le système, identifié par un nom et obtenu par composition d'autres flux et/ou de données. Les flux utilisés pour définir une nouvelle publication sont soit des flux « d'entrée », fournis par le module d'acquisition, soit d'autres publications.

L'utilisateur ROSES peut définir de nouvelles publications et souscrire à ces publications. Une ***souscription*** est une requête continue sur les items d'une publication, spécifiant entre autres une méthode de *notification* (type: par mail, par flux RSS, etc., fréquence de notification, ...).

ROSES offre *un langage déclaratif de spécification de nouvelles publications* de flux par l'utilisateur du système. Une nouvelle publication est définie comme une *vue de composition* (flux virtuel) sur d'autres publications. Il est également possible dans ROSES de matérialiser ces flux virtuels.

Nous présentons ici:

- Le modèle sur lequel ce langage déclaratif s'appuie.
  En dehors du modèle de données et de l'algèbre ROSES (livrable D2.2), la spécification du langage de publication s'appuie sur un certain nombre de choix de modélisation, visant à simplifier la tâche de l'utilisateur et à l'augmenter les possibilités d'optimisation.
- La syntaxe du langage de spécification de nouvelles publications.
- Des exemples d'utilisation pour illustrer les principales caractéristiques de ce langage.

Modèle

a) Types de flux et types d'opérateurs

On peut distinguer deux types de publications ROSES :

- *Matérialisées* : publications qui correspondent à des flux RSS « réels », générés par le module d'acquisition ROSES à partir de sources RSS externes
- *Virtuelles* : publications construites comme des vues de composition sur des publications matérialisées ou virtuelles, à l'aide du langage ROSES présenté ici.

Le langage de publication permet de décrire des compositions de flux (et de données) exprimables avec les opérateurs de l'algèbre ROSES (livrable D2.2 Modèle et Algèbre) :

- Filtrage
- Union
- Fenêtrage
- Jointure / semi-jointure
- Transformation

Ces opérateurs peuvent être classés en deux catégories :

1. *Opérateurs conservatifs*, qui ne créent pas de nouveaux items : filtrage, union, fenêtrage, semi-jointure. On retrouve en sortie de ces opérateurs toujours un sous-ensemble des items en entrée.
2. *Opérateurs altérants*, qui modifient les items d'entrée et créent ainsi de nouveaux items : transformation, jointure (enrichissement).

Les opérateurs conservatifs permettent d'exprimer un grand nombre de requêtes de publication avec des propriétés plus intéressantes en termes d'optimisation de requêtes comparés aux opérateurs altérants. Notamment, les propriétés de commutativité et distributivité permettent une gamme plus large de réécritures équivalentes.

b) Choix de modélisation pour le langage

Les opérateurs altérants sont plus complexes en terme d'exécution et d'optimisation et une étude de différents cas d'utilisation du système ROSES a montré qu'ils sont également moins fréquents que les opérateurs conservatifs. Dans ce contexte, nous avons décidé *d'imposer les contraintes suivantes sur l'utilisation de ces deux types d'opérateurs* pour la définition de de publication :

- Les publications virtuelles (vues) ne sont définies qu'à l'aide d'opérateurs conservatifs, pour faciliter l'optimisation multi-requêtes.
- Les opérateurs altérants peuvent être appliqués aux flux virtuels et matérialisés, mais les flux générés par ces opérateurs seront *matérialisés et* « réinjectés » dans le système au même niveau que les flux d'entrée fournis par le module d'acquisition.

Ces contraintes sont imposés en séparant le langage de publication en deux sous-langages :

- Un sous-langage de définition de publications virtuelles (vues de compositions), basé sur les opérateurs conservatifs.
- Un sous-langage de transformation, basé sur les opérateurs altérants, pour définir des

publications matérialisées, donc produire de nouveaux flux matérialisés.

Un problème particulier à résoudre est celui des *opérateurs de jointure*, qui combinent des caractéristiques conservatives (la semi-jointure) et altérantes (la jointure qui produit un nouvel item à partir de deux items joints). Le choix de modélisation que nous avons fait pour la jointure est le suivant :

- *La jointure se comporte comme une semi-jointure* (opérateur conservatif), donc elle peut être utilisée dans la définition de publications virtuelles.
- A la différence de la semi-jointure classique, on préserve dans chaque item du premier flux une liste d'items correspondants dans le second flux (en fait une liste de références vers les items en question). Un flux ayant subi plusieurs jointures successives conservera dans chaque item une liste pour chaque semi-jointure.
- *Le modèle d'item ROSES est donc étendu* pour inclure cet ensemble de listes. Cet ensemble sera vide pour les items n'ayant pas subi de jointure, notamment les items des flux matérialisés.
- Les listes ne sont utilisables que dans l'opérateur de transformation, qui pourra ainsi produire des items qui « combinent » les contenus des flux joints, comme dans une jointure classique.

La conséquence de ce choix de modélisation est la suivante :

- Le sous-langage pour publications virtuelles sera basé sur les opérateurs de filtrage, union, fenêtrage et jointure.
- Le sous-langage pour publications matérialisés n'exprimera qu'un seul opérateur : la transformation.

c) Choix liés au pouvoir d'expression

Le langage de publication décrit des opérations sur les flux, exprimables par des compositions d'opérateurs algébriques. Néanmoins, un des objectifs de base de ce langage est de fournir *un moyen simple et déclaratif* pour définir de nouvelles publications, facile à maitriser par tout utilisateur. Par conséquent, afin de préserver cette simplicité d'utilisation, le langage de publication peut se limiter à une partie seulement des compositions algébriques possibles et introduire certaines contraintes dans la façon d'exprimer les compositions.

Les principales contraintes sont décrites ci-dessous. Les détails seront discutés au fur et à mesure de la présentation de la syntaxe du langage.

- L'opérateur de fenêtrage n'est utilisé que pour la jointure entre flux : donc tout fenêtrage est suivi par une ou plusieurs de jointures dans les publications. Cela réduit le fenêtrage à son utilisation la plus simple et intuitive, mais couvre la plupart des besoins dans le contexte applicatif ROSES (le même type de contrainte existe dans les langages de flux de données où toutes les fenêtrages doivent être suivi d'un opérateurs d'agrégation).

- Le langage favorise un ordre particulier dans l'expression des opérateurs, qui correspond tout simplement à sa syntaxe. Par exemple, le langage exprime naturellement une union de flux traités par jointure, mais rend plus difficile l'expression d'opérations supplémentaires sur cette union de jointures. On peut parler d'une forme algébrique canonique induite par le langage de publication.
  En réalité, les propriétés de distributivité et commutativité des opérateurs (union, filtrage, jointure avec fenêtrage) font que des expressions complexes puissent être réduites à la forme canonique et ainsi exprimables dans le langage de publication. Néanmoins, il n'est pas

toujours simple pour un utilisateur de faire cette réduction, afin d'exprimer son intention à travers une requête de publication. Il reste dans ce cas possible la voie indirecte, qui consiste à définir le résultat en plusieurs étapes: publications simples, utilisées pour composer des publications plus complexes, avant d'arriver au résultat souhaité.

Langage pour publications virtuelles

Cette section présente les éléments les plus importants pour la compréhension générale du langage. La grammaire complète du langage de publications virtuelles est donnée dans l'annexe.

Notation : on utilise une description syntaxique basée sur des expressions régulières « style DTD » avec ( ) pour grouper, | pour l'alternative et ?, *, + pour les multiplicités. Les mots entre guillemets sont des terminaux de la grammaire, ceux en italique sont des non-terminaux.

La forme générale d'une publication virtuelle est la suivante :

publication := « **create feed** » *feedName* « **as** »
« **return** » *unionFeed*
(« **join** » *joinSpec*)*
(« **where** » *conjCond*)?

- *feedName* est le nom de la nouvelle publication
- *unionFeed* exprime l'ensemble de tous les flux d'entrée qui produisent des items en sortie. Elle peut identifier par des variables des flux individuels ou des unions de flux et exprimer des filtrages sur ces flux ou unions de flux.
- *joinSpec* exprime la jointure d'un des flux de *unionFeed* avec une fenêtre (ou avec une source de données)
- *conjCond* exprime une conjonction de conditions de filtrage sur des flux d'*unionFeed*

Intuitivement, on définit une nouvelle publication virtuelle comme une union de flux d'entrée dans la clause *return*. On peux exprimer des filtrages sur ces flux soit dans la clause *where*, soit directement dans la clause *return*. On peut aussi exprimer des jointures sur ces flux (avec des fenêtres ou avec des sources de données) dans la clause *join*. Enfin, la clause *return* permet aussi de grouper des flux d'entrée par union, afin d'exprimer des filtrages ou des jointures sur ces unions – ce qui permet d'éviter de répéter une même condition de filtrage ou de jointure sur plusieurs flux d'entrée.

a) Clause return

unionFeed := *feed* (« | » *feed*)*

feed := *feedSpec* (« **[** » *predicate* « **]** »)* (« **as** » *variable*)?

feedSpec := *URL* | *feedName* | « **(** » *unionFeed* « **)** »

La clause *return* décrit une union de plusieurs flux (*feed*). Chaque flux provient d'un flux de base (*feedSpec*) éventuellement filtré par des prédicats (*predicate*), le résultat pouvant être identifié par un nom de variable (*variable*).

Chaque flux de base peut être spécifié par une URL de flux RSS/Atom externe (*URL*), par un nom de flux/publication interne (*feedName*) ou par une union de *feed* (*unionFeed*).

Les prédicats de filtrage sont définit par des conjonction de prédicats simples si plusieurs prédicats existent sur un flux, ils sont reliés par une disjonction logique (forme normale disjonctive).

b) Clause join

joinSpec := *windowJoin* | *dataJoin*

windowJoin := « **window** » *windowName windowSpec* « **on** »» *unionFeed* « **with** » *joinCond*

windowSpec := « **last** » *size units*

joinCond := *variable* « **[** » *joinPredicate* « **]** »

dataJoin := « **data** » *windowName query* « **with** » *joinCond*

La clause *join* décrit la jointure d'un des flux de la clause *return* (identifié par une variable) avec une fenêtre (*windowJoin*) ou avec une source de données (*dataJoin*). La fenêtre ou la source de données ont un nom (*windowName*), qui doit être unique dans la requête de publication.

La jointure avec une fenêtre décrit une fenêtre (*windowSpec*) sur un flux (*unionFeed*) et un prédicat de jointure entre un flux de la clause *return* et la fenêtre.

Le prédicat de jointure (*joinPredicate*) est en forme normale disjonctive, comme pour le filtrage et exprime une relation entre le flux de la clause *return* (*variable*) et les items de la fenêtre. Le mot clé *window* permet de faire la différence entre les items de la fenêtre et ceux du flux – par exemple *title* indique le titre d'un item du flux identifié par *variable*, tandis que *window.title* indique le titre d'un item de la fenêtre.

Pour la jointure avec des données, *query* spécifie les données avec lesquelles on fait la jointure, la condition de jointure étant similaire au cas fenêtre.

c) Clause where

conjCond := *cond* (« **and** » *cond*)*

cond := *variable* « **[** » *predicate* « **]** »

La clause *where* exprime des conditions de filtrage sur des flux définis dans la clause *return* et identifiés par des variables. La syntaxe des prédicats est la même que pour le filtrage dans la clause *return*, la clause *where* permet de placer une partie des conditions de filtrage en dehors de la clause *return*, pour éviter qu'elle devienne trop complexe. Cette situation est similaire à celle de XQuery, où les conditions peuvent être exprimées soit sur les chemins XPath, soit dans la clause where.

d) Exemples

- Filtrage : les items du flux "lemonde" dont le titre contient Haiti

  create feed haiti **as**
  return lemonde [title contains "Haiti"]

- Union : les items des flux "lemonde", "libé" et "lefigaro"

  create feed news **as**
  return lemonde | libé | lefigaro

- Filtrage avant union : les items du flux "lequipe" parlant de Domenech dans leur titre et les items du blog "domenechblog"

  create feed domenech **as**
  return lequipe [title contains « Domenech »] | domenechblog

ou

  create feed domenech **as return** lequipe as $e | domenechblog
  where $e.title contains « Domenech »

- Filtrage avant et après union : les items parlant dans le titre de Domenech mais aussi de Blanc.

  create feed domenech_blanc **as**
  return (lequipe [title contains « Domenech »] | domenechblog)[title contains « Blanc »]

ou

  create feed domenech_blanc **as return** (lequipe [title contains « Domenech »] | domenechblog) as $d

  where $d.title contains « Blanc »

- Jointure: les items du flux "lemonde" enrichis avec les items du flux "libé" des 3 derniers jours avec des titres similaires.

  create feed similaire **as**
  return lemonde **as** $m
  join window wlibé **last** 3 **days on** libé **with** $m[title similar window.title]

- Tout ensemble: les items des flux "lemonde" et "libé" enrichis avec les 10 derniers titres du blog de Domenech, ainsi que les items de "lequipe" sur Domenech enrhichi avec les items publiés par "rumeurs" la dernière semaine dans la même catégorie.

  create feed domenech_tout **as**
  return (lemonde | libé) **as** $j | lequipe[title contains « Domenech »] as $e
  join window wdblog **last** 10 **items on** domenechblog
  with $j[description similar window.title]
  join window wrumeurs **last** 7 **days on** rumeurs
  with $e[category = window.category]

Langage pour publications matérialisées

Le langage pour publications matérialisés exprime une transformation des items d'un flux (virtuel ou non) en des items d'un nouveau flux matérialisé.

La transformation porte sur :

- Les composantes de l'item : titre, description, catégorie, etc.
- Si l'item est le résultat d'une ou plusieurs jointures, les listes d'items correspondants.

Elle produit pour chaque item du flux d'entrée un item « matérialisé » dans le flux de sortie, ayant donc les mêmes composantes que l'item d'entrée (avec un contenu modifié), sauf l'ensemble de listes d'items correspondants, qui sera vide.

a) Choix pour la transformation

Afin de simplifier le traitement des différents types de transformations possibles, la transformation dans une publication matérialisée sera spécifiée par *une feuille de style XSLT*. La feuille de style considère en entrée une représentation XML d'item ROSES (incluant les listes d'items joints) et produit en sortie une représentation XML d'item matérialisé. La puissance du langage XSLT permet de traiter une large gamme de transformations, avec agrégation des items joints, etc.

La forme générale d'une publication matérialisée est la suivante :

publication := **« create materialized feed »** *feedName* « **as »**
feedNameIn (« **transformed with »** *XSLTstylesheet*)?

On crée une publication matérialisée *feedName* en transformant les items de la publication d'entrée *feedNameIn* à l'aide de la feuille de style *XSLTstylesheet*. Si la transformation n'est pas spécifiée, on considère la transformation par défaut, qui ne fait qu'éliminer les éventuelles informations de jointure avec des fenêtres ou des sources de données.

b) Exemples

Soit la publication virtuelle *domenech_tout* définie dans les exemples sur le langage de publications virtuelles:

create feed *domenech_tout* **as**
return *(lemonde | libé)* **as** *$j | lequipe[title contains "Domenech"]* as *$e*
join window *wdblog* **last** 10 **items on** *domenechblog*
with *$j[description* similar *window.title]*
join window *wrumeurs* **last** 7 **days on** *rumeurs*
with *$e[category = window.category]*

On peut matérialiser ce flux virtuel en lui appliquant aussi une transformation *domenech.xsl*, à l'aide de la commande:

create materialized feed *domenech_mat* **as**
domenech_tout transformed with "domenech.xsl"

Résultat avant transformation :

```
<feed xmlns='roses.syndicate.example' name='domenech_tout'>
  <item>
    <title>...</title>
    <description>...</description>
    <contents>...</contents>
   ...
    <window name='domenech_tout_wdblog>
        <item>...</item>
        <item>...</item>
           ....
    </window>
     <window name='domenech_tout_wrumeurs>
        <item>...</item>
        <item>...</item>
           ....
    </window>
  </item>
  <item>...  </item>
 ...
</feed>
```

Chaque item contient en plus de ses champs habituels, un sous-élément *window* pour chaque fenêtre de jointure de la publication (et chaque fenêtre de jointure des publications d'entrée). L'attribut *name* identifie d'une manière unique la fenêtre, il est composé du nom de la publication concaténé au nom de la fenêtre dans la publication. Les sous-éléments *item* de *window* sont les items de la fenêtre qui satisfont la condition de jointure avec l'item courant du flux.

La règle par défaut efface toutes les informations liées à la jointure. L'effet est le même que si toutes les jointures étaient des semi-jointures :

```
<xsl:template match='window'/>
```

Autres règles utiles dans les feuilles de style de transformation:

- Pour effacer un élément particulier, il suffit d'ajouter une règle d'effacement (effacer les descriptions) :

  ```
  <xsl:template match='description'/>
  ```

- Pour agréger des items provenant de jointures: exemple de règle qui rajoute à la description un commentaire avec le nombre d'items correspondants dans la fenêtre *wdblog* (donc le nombre de commentaires récents sur le *domenechblog* relatifs à l'item courant de l'Equipe).

  ```
  <xsl:template match='description'>
     <xsl:copy>
          <xsl:value-of select='.'/> (<xsl:value-of select = 'count (../window[@name=
  "domenech_tout_wdblog" ]/item)'/> commentaires récents sur le blog de Doménech)
     </xsl:copy>
  </xsl:template>
  ```

- Pour effacer les items provenant d'un fenêtrage particulier, il suffit d'identifier la fenêtre à effacer (effacer les rumeurs) :

    <xsl:template match='window[@name="domenech_tout_wrumeurs"]/>

Annexe : grammaire du langage de publications virtuelles

Non-terminaux

```
query     :=       "create" "feed" feedName "as" "return" union ( joinClause )*
( whereClause )? <EOF>

feedName  :=       <NAME>

union     :=       ( feed ( <UNION> feed )* )

feed      :=       ( string | feedName | "(" union ")" ) ( "[" predicate "]" )*
( "as" variable )?

string    :=       <STRING>

// priorites : OR < AND < NOT < ()
predicate :=       disjunction

disjunction        :=       ( conjunction ( "or" conjunction )* )

conjunction        :=       ( literal ( "and" literal )* )

literal   :=       atom
                   |        "not" atom

atom      :=       contains
                   |        "(" predicate ")"

contains  :=       attribute "contains" string

attribute :=       <ATTRIBUTE>

variable  :=       <VARIABLE>

joinClause         :=       "join" "window" windowName windowPredicate "on"
union "with" joinSatisfies

windowName         :=       <NAME>

windowPredicate    :=       "last" integer unit

integer   :=       <INTEGER>

unit      :=       <UNIT>

joinSatisfies      :=       variable "[" joinPredicate "]"
// meme q predicat selection sauf pour les atoms!

joinPredicate      :=       joinDisjunction

joinDisjunction    :=       ( joinConjunction ( "or" joinConjunction )* )

joinConjunction    :=       ( joinLiteral ( "and" joinLiteral )* )

joinLiteral        :=       joinAtom
                   |        "not" joinAtom

joinAtom  :=       similar
                   |        "(" joinPredicate ")"

similar   :=       ( attribute "similar" "window" "." attribute | "window" "."
attribute "similar" attribute )

whereClause        :=       "where" satisfies ( "and" satisfies )*

satisfies :=       variable "[" predicate "]"
```

Terminaux

```
<CREATE: "create">
<FEED: "feed">
<AS: "as">
<RETURN: "return">
<JOIN: "join">
<WINDOW: "window">
<LAST: "last">
<ON: "on">
<WITH: "with">
<WHERE: "where">
<OR: "or">
<AND: "and">
<NOT: "not">
<CONTAINS: "contains">
<SIMILAR: "similar">
<ATTRIBUTE: "title" | "description" | "author">
<UNION: "union" | "|">
<UNIT: "items" | "seconds" | "minutes" | "hours" | "days" | "weeks" | "months">
<NAME: ["a"-"z","_"] (["a"-"z","_","0"-"9"])*>
<STRING: "\"" (~["\""])+ "\"">
<VARIABLE: "$" ["a"-"z","_"] (["a"-"z","_","0"-"9"])*>
<INTEGER: (["0"-"9"])+>
```