
World Wide Web: Internet and Web Information Systems, 5, 325–366, 2002

Dynamic Restructuring of E-Catalog Communities Based on User Interaction Patterns

HYE-YOUNG PAIK, BOUALEM BENATALLAH and RACHID HAMADI
{hpaik,boualem,rhamadi}@cse.unsw.edu.au
School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia

Abstract

Since e-catalogs are dynamic, autonomous, and heterogeneous, the integration of a potentially large number of dynamic e-catalogs is a delicate and time-consuming task. In this paper, we describe the design and the implementation of a system through which existing on-line product catalogs can be integrated, and the resulting integrated catalogs can be continuously adapted and personalized within a dynamic environment. The integration framework originates from a previous project on integration of Web data, called WebFINDIT. Using the framework, we propose a methodology for adaptation of integrated catalogs based on the observation of customers' interaction patterns.

Keywords: e-catalog, e-catalog community, user interaction patterns, catalog structural adaptation, catalog personalization

1. Introduction

In recent years, integration of e-catalogs has gained considerable momentum because of the emergence of online shopping portals, increasing demand for information exchange between trading partners, the prevalence of mergers and acquisitions [19,29]. In approaches that address the problem of e-catalog organization and integration, a product catalog is usually structured in a category-based hierarchy [13,19]. Catalogs are designed in a "one-view-fits-all" fashion, by a system designer who has *a priori* expectations for how catalogs will be "explored" by customers. However, the customers may have different expectations. Therefore, it is necessary to take into account how the customers are using the catalogs to continuously minimize the gap between expectations of the system designer and customers. For example, in a catalog for computer parts, where it is repeatedly observed that many users always use product category `RAM` immediately after using category `CPU`. If the administrator merges the two categories and creates a new category `CPU&RAM`, users only need to visit this new category once for information on both products.

In this paper, we describe the design and the implementation of a system, called WebCatalog^{Pers} [20], through which existing online product catalogs can be integrated, and the resulting integrated catalogs can be continuously adapted and restructured within a dynamic environment. The catalog integration framework used in this paper originates from a previous project on integration of Web data, called WebFINDIT [6,7].[1] Based on this

framework, we propose a usage-centric technique for transforming catalog organization. The objective is to *continuously* improve the organization of catalogs by being responsive to the ways customers navigate them when searching for products. The proposed approach offers the following features:

- *Catalog navigation and access model.* This model provides a set of actions called catalog interaction actions that users would perform while accessing catalogs. The actions are tracked and analyzed to find any pattern in users' interaction behavior.
- *Catalog transformation operations.* These operations are used to transform the structure and organization of catalogs (e.g., splitting a catalog community, merging catalog communities).
- *Predefined sequences of catalog interaction actions.* These sequences represent preidentified interaction patterns of users. They can be considered as heuristics for catalog transformation. Discovery of these patterns helps administrators (e.g., communities owners) decide what kind of transformations would be desirable to improve the organization of catalogs. Transformations of a catalog over time result in offering improved alternative of its organization based on user interaction patterns.

Preliminary work on the proposed approach has been presented in [20]. The remainder of this paper is organized as follows. Section 2 overviews the design of WebCatalog$^{\text{Pers}}$. Section 3 presents a formal model for integrated catalogs and user interaction actions. The catalog reorganization operations are introduced in Section 4. Section 5 explains the concept of *Predefined Interaction Sequences* (PIS) which is used to discover users' behavior patterns. In Section 6, we discuss handling of situations where conflict and consolidation relationships exist between users' behavior patterns. Implementation overview is presented in Section 7. Section 8 presents the results of simulation studies. Section 9 discusses related work and, finally, Section 10 gives concluding remarks.

2. WebCatalog$^{\text{Pers}}$: Design overview

In this section, we give the intuition behind the main concepts that are used in WebCatalog$^{\text{Pers}}$, namely, product catalogs and catalog communities. The formalization of these concepts will be presented in the next section.

2.1. Catalog communities

A catalog community is specialized to a single area of products (e.g., a community for `Motherboards`, or `Printers` etc.). It contains the description of domain-specific product attributes and terms for interacting within a catalog community, and individual product catalogs which are members of the community (e.g., a product catalog for `Canon Printers` for community `Printers`). We illustrate catalog communities with an example from the domain of *computer parts and related services* (see Figure 1).

There are two types of relationships defined between catalog communities: *SubCommunityOf* and *PeerCommunityOf*. SubCommunityOf relationships represent *specialization*

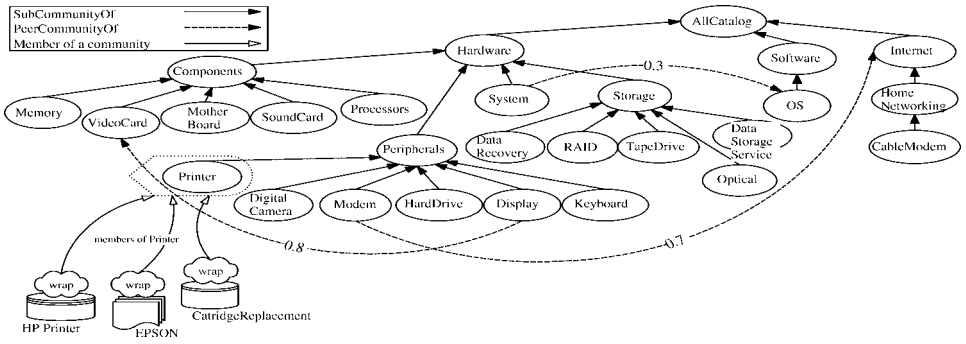

Figure 1. eCatalogs-Net: Organizing catalog communities.

between domains of two catalog communities. For example, community `Printer` is a sub-community of `Peripherals`.[2] PeerCommunityOf represents a relationship between two communities in which one community can be an alternative of the other community. In Figure 1, community `Display` is a peer community of `VideoCard`. It should be noted that we do not assume that the opposite (i.e., `VideoCard` is a peer community of `Display`) systematically holds. A weight (i.e., a real value between 0 and 1) is attached to each PeerCommunityOf relationship, which represents a degree of relevancy of the relationship. PeerCommunityOf relationships can be viewed as a referral mechanism in that when the user cannot find (or is not satisfied with) information from a catalog community, she/he can refer to other communities that the catalog community considers as its peers. We call this organization of catalog communities *eCatalogs-Net*. `AllCatalog` is a special community to which a community that is not a sub-community of any catalog community is related via SubCommunityOf relationship. It should be noted that communities in the eCatalogs-Net can forward requests to each other.

Our observation shows that the most common form of representing a product `p` in a catalog is using a set of attributes $A = \{a_1, a_2, \ldots, a_n\}$, where each a_i is an attribute that describes properties of `p` [19,29,30]. In our approach, a creator of community defines the attributes suitable for commonly describing underlying product catalogs, which ultimately forms the schema of the community. Therefore, each catalog community has a set of attributes which represents the product catalogs in the community. We refer to the set of attributes as *community product attributes* (*community attributes*, for short). For example, community attributes of a catalog community of `Printers` are described as follows:

```
CatalogCommunity Printers {
   CommProdAttr{(Manufacturer_Name, String),(Print_Resolution, String),
                (Body_Colour, String),(Unit_Price, Float),
                (LeadTime_Required, Integer),(Full_Description, String)}}
```

In the above example, `Manufacturer_Name` is an attribute and `String` is a type of the attribute. Once community attributes have been identified, it is left to the product catalog providers to map their local attributes (i.e., the attributes that describes products in their catalogs) to the community attributes so that their catalog can be queried using the

attributes of the community. More details about catalog registration will be discussed in Section 2.2. It should be noted that it is also possible to use a product catalog description standard (e.g., xCBL, Dublin Core Metadata Initiative) [2,8,11,28]. Although a presentation model for e-catalog is an important issue, further discussion is beyond the scope of this paper.

2.2. *Catalog registration*

A product catalog provider registers his/her local product catalog with one or more catalog communities to become a member. To explain the registration process, let us assume that there is a product catalog `AceHardware`, whose product types are printers and scanners. The `AceHardware` will be known to a catalog community by providing: (i) a wrapper, (ii) an exported interface, and (iii) a mapping between exported interface and community attributes. Let us assume that `AceHardware` wants to become a member of the community `Printers`. Then, it uses the following statement to register with `Printers`:

```
Registration Source:AceHardware, Target:Printer {
   Wrapper "db.catalog.acehardware.com.au/~webcatalog/OracleWrap";
   Exported Interface PrinterInt;
   Mapping AceHardwareToPrinters;}
```

Users may use a community to express queries that require extracting and combining product attributes from multiple underlying product catalogs. For example, being able to compare prices or descriptions of products is crucial for users to make informed choices. We refer to these types of queries as *global queries*. A wrapper translates global queries (i.e., queries expressed using the community attributes) to local queries (i.e., queries expressed using the local product catalog's native query language). The outputs produced in response to local queries are translated back into the format used by WebCatalog^{Pers}. The exported interface defines the product attributes to query product information at the local catalog. For example, `AceHardware` exports the following interface:

```
Interface PrinterInt {
   LocalAttribute String Brand;
   LocalAttribute String Description;
   LocalAttribute Float Retail_Price;}
```

Global querying is achieved by using community attributes. Community product attributes do not directly correspond to product attributes of local product catalogs. Therefore, when a product catalog is registered with a community, the catalog provider should also, define mapping between local product attributes and community attributes. We call this mapping *Source–Community mapping*. For example, the mapping between `AceHardware` and `Printers` can be described as follows:

```
AceHardwareToPrinters Source:AceHardware, Target:Printers {
   Attribute String Brand IS String Printers.Manufacturer_Name;
   Attribute String Description IS String Printers.Full_Description;
   Attribute Float Retail_Price IS Float Printers.Unit_Price;}
```

In the example, the attribute `Brand` in `Ace Hardware` is related (i.e., mapped) to the attribute `Manufacturer_Name` in `Printers`, `Description` to `Full_Description`. It is possible that `AceHardware` becomes a member of a different community (e.g., `Computer Accessories`) whose attributes are different from `Printers`. Naturally, the Source–Community mapping will be separately defined. Catalog registration may concern only a subset of the attributes of a community. Finally, a community can be registered with another community. By doing so, the members of the first community become members of the second community as well. For example, the catalog `TFT Monitors` and `HighTech Monitors` are registered with the community `Monitors`, which itself is registered with the community `ComputerDisplayUnits`.

2.3. Searching and querying product information

Communities in WebCatalog^{Pers} are presented to users by displaying direct neighbors known via the relationships. Users in WebCatalog^{Pers} will typically be engaged in two-step product information-seeking activity: (i) browsing communities for product catalog location (e.g., get communities that are relevant to selling laptops) and (ii) querying selected communities or catalogs for products information (e.g., compare product prices). Users would have a specific task or a goal to achieve in mind (e.g., product items that wish to purchase, a category of products they want investigate) when using product catalogs. We assume that they use the following strategy:[3]

1. Start at the root (i.e., `AllCatalog`), or at a specific community (if they know the location of the catalog community).
2. While (current community C is not the target community T) do
   (a) If any of the SubCommunityOf relationships of C seems likely to lead to T, follow the relationship that appears most likely to lead to T.
   (b) Else, if any of the PeerCommunityOf relationships of C seems likely to lead to T, follow the relationship that appears most likely to lead to T.
   (c) Else, either backtrack and follow SuperCommunityOf relationship of C, or give up searching.

Once the user has reached a community she/he was looking for, she/he will submit a query to it. If the user ends up in the same community again in step 2(a) or 2(b), she/he will follow a different relationship, since her/his reasoning of which relationship is likely to lead to the target is changed by then.

3. Modeling catalog communities and user interaction

In this section, we introduce a formal model for representing product catalogs, communities, and eCatalogs-Net. Then, we identify a set of permissible actions that users can perform when interacting with eCatalogs-Net.

3.1. *eCatalogs-Net*

We give the definition of eCatalogs-Net after formally introducing the definitions of product catalog and catalog community.

Definition 1 (Product catalog). A product catalog is a tuple $P = (\texttt{NameP}, \texttt{GeneralAttr}, \texttt{ProductAttr})$ where

- `NameP` is the name of the product catalog `P`,
- `GeneralAttr` is a set of name-value pairs (`a`, `v`), where `a` is an attribute and `v` is a value of `a` (e.g., (URL, "http://www.cse.unsw.edu.au/~LinuxSoftware")), and
- `ProductAttr` is a set of attribute–type pairs (`att`, `type`) used to describe products (e.g., (`Product_Name`, `String`)).

Definition 2 (Catalog community). A catalog community C is a tuple $C = (\texttt{NameC}, \texttt{GeneralInfo}, \texttt{CommProdAttr}, \texttt{Members})$ where

- `NameC` is the name of the community C,
- `GeneralInfo` is a set of pairs (`p`, `v`), where `p` is a property of the community and `v` is a value of `p` (e.g., (`Domain`,`"CD Writers"`)),
- `CommProdAttr` is a set of attribute–type pairs (`att`, `type`), where `att` is a community product attribute and `type` is the type of `att` (e.g., (`Model_Numer`, `Integer`)),
- `Members` is a set of members. A member can be either a product catalog or another catalog community and is defined as a pair (`mid`, `map`) where `mid` represents the identifier of the member and `map` contains the Source–Community mapping.

Definition 3 (eCatalogs-Net). An eCatalogs-Net is a labelled directed graph $G = (N, E_1, E_2, W, \ell)$ where

- N is a finite set of nodes; a single node represents a catalog community,
- $E_1 \subseteq N \times N$ is a finite set of directed edges (representing SubCommunityOf relationships),
- $E_2 \subseteq N \times N$ is a finite set of directed edges (representing PeerCommunityOf relationships),
- $W : E_2 \rightarrow [0, 1]$ is a weighting function (initially each edge in E_2 receives a neutral weight of 0.5), and
- $\ell : N \rightarrow \mathcal{C}$ is a naming function where $\mathcal{C}$ is a set of catalog communities names.

3.2. *Permissible user actions*

The permissible actions, noted $\mathcal{A}$, for exploring eCatalogs-Net are listed in Table 1. By modeling user interaction actions, the system can capture them for future use. It should be noted that the focus of this paper is not on specifying transactional operations (e.g., or-

Table 1. Permissible user actions $\mathcal{A}$ in eCatalogs-Net

Action name	Description
`NavigateToSub(catalogCommunity c)`	Invoked when the user goes, from the current catalog community, to one of its sub-communities, namely, *c*.
`NavigateToSuper()`	Invoked when the user goes, from the current catalog community, to its super-community.
`NavigateToPeer(catalogCommunity c)`	Invoked when the user goes, from the current catalog community, to one of its peer communities *c*.
`SelectBookmark(Bookmark b)`	Invoked when the user selects the bookmark *b* defined at the current community. WebCatalog$^{\text{Pers}}$ then automatically performs the sequence of actions recorded in the bookmark (refer to Section 3.3).
`LeaveCatalogCommunity()`	Invoked when the user leaves the current catalog community. The user is taken to the default community `AllCatalog`.
`ShowMembers(Constraint const)`	Invoked when the user request to show the members of the current catalog community satisfying the constraint `const`.
`SubmitQuery(Query q)`	Invoked when the user submits the query `q` to the current catalog community. It could be a global query which uses the community's community attributes, or a source query which concerns one member of the current community (i.e., uses the exported attributes of the associated member).

dering or payment for products). We assume that it is up to the local product catalog supplier to process such operations via interfaces exported to WebCatalog$^{\text{Pers}}$. Detailed description on provisioning operations in the context of Web services is presented in [3].

3.3. WebCatalog$^{\text{Pers}}$ *bookmarks*

WebCatalog$^{\text{Pers}}$ uses the concept of bookmarks. Bookmarks in WebCatalog$^{\text{Pers}}$ are different from conventional Web browsers' bookmarks which usually store a list of URLs. In WebCatalog$^{\text{Pers}}$, a single bookmark is a recorded sequence of user actions. When a system administrator discovers that a certain sequence of actions is performed frequently by the users, she/he may record such actions as a bookmark making it available so that the same sequence of actions can be easily repeated later by users. A bookmark is formally defined as follows:

Definition 4 (Bookmark). A bookmark is a pair $B = (\texttt{IdB}, \langle a_1, a_2, \ldots, a_n \rangle)$ where

- `IdB` is the identifier of the bookmark B,
- $\langle a_1, a_2, \ldots, a_n \rangle$ is an ordered sequence of permissible actions where $a_i \in \mathcal{A}$ $(i = 1, \ldots, n)$.

After a new community is defined, a set of bookmarks can be associated with it. A bookmark may become invalid when a community referenced in any action of the bookmark

is changed (e.g., deleted, split etc., see Section 4). For example, assume that there is a bookmark which contains an action `NavigateToSub(CableModem)`. The action navigates from the "*current*" community, down to its sub-community `CableModem`. Assume also that later, the community `CableModem` is merged with `HomeNetworking` and given a new name. This means that the bookmark that contains an action which references `CableModem` is now *invalid* because `CableModem` no longer exists. Therefore, after a restructuring operation, we check for any invalidity that may have been caused by the operation on bookmarks to make sure that all bookmarks are still valid. Informally, a bookmark is *valid*, if: (i) all communities referenced in any action of a bookmark exist in eCatalogs-Net and have not been moved to a new location and (ii) the attributes of a community c used by a global query in `SubmitQuery` or `ShowMembers` exist in c. The first task in checking the validity of a bookmark is to identify all communities referenced from actions in the bookmark. All communities thus referenced are identified by a *bookmark path*.

Definition 5 (Bookmark path). A bookmark path BP of a bookmark $B = (\texttt{IdB}, \langle a_1, a_2, \ldots, a_n \rangle)$ in community c of the eCatalogs-Net $G = (N, E_1, E_2, W, \ell)$ is a sequence of nodes $\text{BP} = (c_0, c_1, c_2, \ldots, c_n)$ where

- $c_i \in N$ for $i = 0, \ldots, n$,
- $c_0 = c$ is the community from which the recording of bookmark B started,
- c_i, for $i = 1, \ldots, n$, is the target community[4] of the action a_i,
- for $i = 1, \ldots, n$ if $c_{i-1} \neq c_i$ then $(c_{i-1}, c_i) \in E_1 \cup E_1^{-1} \cup E_2$ (i.e., a_i is one of the navigation actions `NavigateToSuper`, `NavigateToSub`, or `NavigateToPeer`), or $c_i =$ `AllCatalog` if a_i is `LeaveCatalogCommunity`,
- for $i = 1, \ldots, n$ if $c_{i-1} = c_i$ then a_i is one of non-navigation actions `SubmitQuery`, `ShowMembers`, or `SelectBookmark`.

For example, the following bookmark B is defined in community `BackupDevices`:

$B =$ (`bmkId001`, ⟨`NavigateToSub(ZipDrive)`,
`NavigateToPeer(PortableHardDrives)`,
`SubmitQuery("Select Price,`
`ModelName From PortableHardDrives")`⟩).

Note that `ZipDrive` is a sub-community of `BackupDevices`. The associated bookmark path BP of the bookmark B is

BP = (`BackupDevices`, `ZipDrive`, `PortableHardDrives`,
`PortableHardDrives`).

The recording started at `BackupDevices`, and then from there the user navigated to `ZipDrive`, then again to `PortableHardDrives` before finally submitting a query to `PortableHardDrives`. Using this bookmark path, a valid bookmark is defined as follows:

Definition 6 (Valid bookmark). A bookmark $B = (\texttt{IdB}, \langle a_1, a_2, \ldots, a_n\rangle)$ in eCatalogs-Net $G = (N, E_1, E_2, W, \ell)$ is *valid* if and only if its bookmark path $BP = (c_0, c_1, c_2, \ldots, c_n)$ satisfies the following conditions:

1. $c_i \in N$ for $i = 0, \ldots, n$, i.e., each community in BP must exist in eCatalogs-Net.
2. For each a_i such that $c_i \neq c_{i+1}$ $(i = 0, \ldots, n-1)$, $(c_i, c_{i+1}) \in E_1 \cup E_1^{-1} \cup E_2$ or $c_{i+1} = \texttt{AllCatalog}$. c_i and c_{i+1} are related by one of the following relationships; SubCommunityOf, SuperCommunityOf and PeerCommunityOf in eCatalogs-Net.
3. For each a_i such that $c_i = c_{i+1}$ $(i = 0, \ldots, n-1)$, the action a_i is a valid non-navigation action, that is:
   - if $a_i = \texttt{SubmitQuery(q)}$ then the community attributes in q must exist in c_i,
   - if $a_i = \texttt{ShowMembers(cons)}$ then the community attributes in cons must exist in c_i,
   - if $a_i = \texttt{SelectBookmark(b)}$ then b must exist in c_i.

For an eCatalogs-Net to be *consistent*, it must satisfy the following conditions: (1) each catalog community has a unique name, (2) each catalog community cannot have SubCommunityOf relationship with neither its ancestors nor with its descendants (other than its super-community), (3) each catalog community cannot have PeerCommunityOf relationship with its ancestors nor with its descendants. Note that *ancestors* and *descendants* of a community are defined w.r.t. SubCommunityOf relationships, and (4) all bookmarks are valid. Those conditions are, respectively, formally stated in the following definition.

Definition 7 (Consistent eCatalogs-Net). The eCatalogs-Net $G = (N, E_1, E_2, W, \ell)$ is consistent if and only if the following conditions are satisfied:

- The naming function ℓ is injective (that is, there will not be two communities with the same name),
- The graph $G_1' = (N, E_1^{-1}, \ell)$ (generated from the subgraph $G_1 = (N, E_1, \ell)$ of G by inverting the edges) is a tree. The root of the tree is AllCatalog, and
- $E_2 \cap (E_1 \cup E_1^{-1})^+ = \emptyset$ (where E^+ denotes the transitive closure of E, i.e., $(i, j) \in E^+$ iff there is a directed path from i to j in E).
- All bookmarks defined in catalog communities of eCatalogs-Net are valid.

3.4. *Capturing user behavior*

To analyze user behavior, we need to record all actions that have occurred during the time the user interacted with the system. Moreover, after recording all user actions, we need to transform the collection of raw data into the format which is suited by the system to carry out required processing.

Every time a user invokes one of the permissible actions at a catalog community, WebCatalog$^{\text{Pers}}$ keeps that event in the system log file. Each element in the log file contains the name of the action, a user identifier (UID), a time stamp (TS), and parameters of the action. For example, the first log file entry below shows that the user, whose UID is 987,

was at Hardware catalog community, then navigated down to its sub-community Modem (see Figure 1), on 10/08/2001 13:05:40 system time, etc.:

```
(NavigateToSub, UID=987, TS="10:08:2001:13:05:40",
   C_FROM="Hardware", C_TO="Modem")
(NavigateToSuper, UID=811, TS="10:08:2001:13:05:42",
   C_FROM="IBM", C_TO="Retailers")
```

To make use of log data when searching interaction patterns (see Section 5), we make some transformations on the log file. We first, *sessionize* the log file. We define a session as follows:

Definition 8 (Session). A session is an ordered sequence of actions performed by a single user, where the time difference between any two consecutive actions in the list should be within a preselected time threshold, $T_{\text{threshold}}$.

After log file sessionizing, we extract all community attributes selected by global queries. The extraction process converts a global query, which is contained in the Query parameter of SubmitQuery actions, into a list of community attributes. The generated list is stored as a value of a special attribute called GLB_QUERY_ATTR. This is done for the purpose of analyzing usage pattern of community attributes by users. In the following, we show an example of a session transformed from a log file. All actions belonging to the same user are identified[5] and global query attributes are extracted.

```
(Session="Start")
   (NavigateToSub, C_FROM="AllCatalog", C_TO="Hardware")
   (NavigateToSub, C_FROM="Hardware", C_TO="Systems")
   (NavigateToSuper, C_FROM="Systems", C_TO=""Hardware")
   (NavigateToSub, C_FROM="Hardware", C_TO="Components")
   (SubmitQuery, C_SUBMITTO="Components",
      GLB_QUERY_ATTR="Manufacturer, Description, Price")
(Session="End")
```

4. Restructuring eCatalogs-Net

We now describe a set of restructuring operations on eCatalogs-Net. These operations are used, for example, to change the relationships between catalog communities (e.g., update a PeerCommunityOf relationship), remove a catalog community, or merge catalog communities, etc. They can be performed at an administrator's own discretion. In the next section, we will introduce predefined interaction sequences which provide means to observe the user's interaction patterns. The observation will help decide which operation to perform in order to improve the organization of the eCatalogs-Net. An operation is applied to a consistent eCatalogs-Net $G = (N, E_1, E_2, W, \ell)$ and produces a consistent eCatalogs-Net $G' = (N', E_1', E_2', W', \ell')$. We first list the operations in Table 2 and then give details in the following subsections.

Table 2. eCatalogs-Net restructuring operations

Primitive operations	Description
`setCatCommName(catalogCommunity c, String n)`	Set the name of catalog community c to n.
`addPeer(catalogCommunity` c_i`,` `catalogCommunity` c_j`)`	Add a PeerCommunityOf relationship from c_i to c_j with default weight of 0.5.
`delPeer(catalogCommunity` c_i`,` `catalogCommunity` c_j`)`	Delete the PeerCommunityOf relationship from c_i to c_j.
`updatePeer(catalogCommunity` c_i`,` `catalogCommunity` c_j`, Weight w)`	Update the weight of the PeerCommunityOf relationship from c_i to c_i, by w.
`addSub(catalogCommunity` c_i`,` `catalogCommunity` c_i`)`	Add a SubCommunityOf relationship from c_i to c_j.
`delSub(catalogCommunity` c_i`,` `catalogCommunity` c_j`)`	Delete the SubCommunityOf relationship from c_i to c_j.
`createCatComm (String n, GeneralInfo gi, Members m, CommProdAttr gs)`	Create a new catalog community with `Name n, GeneralInfo gi, Members m, CommProdAttr gs`.
`superCatComm(catalogCommunity c)`	Return the super-catalog community of c.
`subCatComm(catalogCommunity c)`	Return direct sub-catalogs communities of c.
`indSubCatComm(catalogCommunity c)`	Return direct and indirect sub-communities of c.
`addBookmark(catalogCommunity c, BookmarkId b, SeqActions sa)`	Add a bookmark `(b,sa)` to `Bookmarks` of c.
`delBookmark(catalogCommunity c, BookmarkId b)`	Remove the bookmark `b` from `Bookmarks` of c.
`updateBookmark (catalogCommunity c, BookmarkID b, catalogCommunity` c_{old}`, catalogCommunity` c_{new}`)`	Replace every occurrence of c_{old} with c_{new} in a bookmark path associated with `b` in community c.
High-level operations	**Description**
`delCatComm(catalogCommunity c)`	Delete the catalog community c.
`moveCatComm(catalogCommunity` c_i`,` `catalogCommunity` c_j`)`	Move the catalog community c_i to its new super-community c_j.
`mergeCatComm(catalogCommunity` c_i`,` `catalogCommunity` c_j`, String n)`	Merge two existing catalog communities c_i and c_j and set the name of new catalog community to n.
`splitCatComm(catalogCommunity c, GeneralInfo gic, String n, GeneralInfo gi, Query q, CommProdAttr cpa, setOfCommunities sub)`	Split the catalog community c into two separate catalog communities.

4.1. Primitive operations

In the following, we introduce primitive operations in details. For each operation introduced here, we show signature, preconditions and the effects.

`setCatCommName(catalogCommunity c, String n)`

Precondition: $c \in N \wedge \forall c' \in N - \{c\}, \ell(c') \neq n$.
Effects: `c.NameC` is set to `n` (i.e., `c.NameC = n`).

`addPeer(catalogCommunity ci, catalogCommunity cj)`

Precondition: there should be no existing PeerCommunityOf relationship from c_i to c_j, and c_i, c_j are not related by a super, or sub-community relationships (i.e., $(c_i, c_j) \notin E_2 \wedge c_i, c_j \notin (E_1 \cup E_1^{-1})^+$).
Effects: A new edge from c_i to c_j is created. We get a graph $G' = (N, E_1, E_2 \cup \{(c_i, c_j)\}, \ell, W \cup \{((c_i, c_j), 0.5)\})$.

`delPeer(catalogCommunity ci, catalogCommunity cj)`

Precondition: $(c_i, c_j) \in E_2$.
Effects: The existing edge from c_i to c_j is removed. The output graph is $G' = (N, E_1, E_2 - \{(c_i, c_j)\}, \ell, W - \{(c_i, c_j), W(c_i, c_j)\})$.

`updatePeer(catalogCommunity ci, catalogCommunity cj, Weight w)`

Precondition: $(c_i, c_j) \in E_2 \wedge -1 \leqslant w \leqslant 1$.
Effects: The weight of PeerCommunityOf relationship from c_i to c_j, $W(c_i, c_j)$ is updated. If $W(c_i, c_j) + w > 1$ then $W'(c_i, c_j) = 1$, if $W(c_i, c_j) + w < 0$ then $W(c_i, c_j) = 0$, and otherwise $W'(c_i, c_j) = W(c_i, c_j) + w$. We get a graph $G' = (N, E_1, E_2, W \cup \{W'(c_i, c_j)\}, \ell)$.

`addSub(catalogCommunity ci, catalogCommunity cj)`

Precondition: $(c_i, c_j) \notin (E_1 \cup E_1^{-1})^+ \cup E_2$.
Effects: A new SubCommunityOf relationship from c_i to c_j is created. We get a graph $G' = (N, E_1 \cup \{(c_i, c_j)\}, E_2, W, \ell)$.

`delSub(catalogCommunity ci, catalogCommunity cj)`

Precondition: $(c_i, c_j) \in E_1$.
Effects: The SubCommunityOf relationship from c_i to c_j is removed. We get a graph $G' = (N, E_1 - \{(c_i, c_j)\}, E_2, W, \ell)$.

`createCatComm(String n, GeneralInfo gi, Members m, CommProdAttr gs)`

Precondition: $\forall c \in N, \ell(c) \neq n$.
Effects: A new catalog community is created. The new catalog community named n does not have a relationship to any other catalog community yet (i.e., must be followed by `addSub()` operation).

`superCatComm(catalogCommunity c)`

Precondition: $c \in N - \{\texttt{AllCatalog}\}$.
Effects: A catalog community which has SuperCommunityOf relationship with c is returned (i.e., return c_1 such that $(c, c_1) \in E_1$).

`subCatComm(catalogCommunity c)`

Precondition: $c \in N$.
Effects: A set of catalog communities which, directly, have SubCommunityOf relationship with c are returned (direct sub-communities) (i.e., return all $c_1 \in N$, such that $(c_1, c) \in E_1$).

`indSubCatComm(catalogCommunity c)`

Precondition: $c \in N$.
Effects: A set of catalog communities which, directly or indirectly, have SubCommunityOf relationship with c are returned (indirect sub communities) (i.e., return all $c_1 \in N$, such that $(c_1, c) \in E_1^+$).

```
addBookmark(catalogCommunity c, BookmarkId b,
   SeqActions sa)
```

Precondition: $c \in N \wedge \forall(\texttt{id}, \texttt{seqact}) \in \texttt{c.Bookmarks}, b \neq \texttt{id}$.
Effect: $\texttt{c.Bookmarks} = \texttt{c.Bookmarks} \cup \{(\texttt{b}, \texttt{sa})\}$.

`delBookmark(catalogCommunity c, BookmarkId b)`[6]

Precondition: $c \in N \wedge (b, \bullet) \in \texttt{c.Bookmarks}$.
Effect: $\texttt{c.Bookmarks} = \texttt{c.Bookmarks} - \{(b, \bullet)\}$.

```
updateBookmark(catalogCommunity c, BookmarkID b,
   catalogCommunity c_old, catalogCommunity c_new)
```

Let BP(b) be a function that produces the bookmark path from a bookmark id b.
Precondition: $c, c_{\text{old}}, c_{\text{new}} \in N \wedge (b, \bullet) \in \texttt{c.Bookmarks} \wedge c_{\text{old}} \in \text{BP}(b)$.
Effect: $\forall c_i \in \text{BP}(b)$, if $c_i = c_{\text{old}}$ then $c_i = c_{\text{new}}$.

4.2. High-level operations

High level operations are more complex operations such as deleting, moving, splitting, or merging catalog communities. Each high level operation is defined as a sequence of primitive operations.

4.2.1. Deleting a community. The operation `delCatComm()` removes a catalog community from the eCatalogs-Net. It is used, e.g., when a catalog community becomes obsolete (e.g., has no useful existence inside the eCatalogs-Net). Figure 2 illustrates the effects of this operation (deletion of community E). All incoming and outgoing edges of E (i.e., super- and sub-community, peer-community relationships) have to be removed and bookmarks also have to be checked for validity.

`delCatComm (catalogCommunity c)`
Precondition: $c \in N - \{\texttt{AllCatalog}\}$.

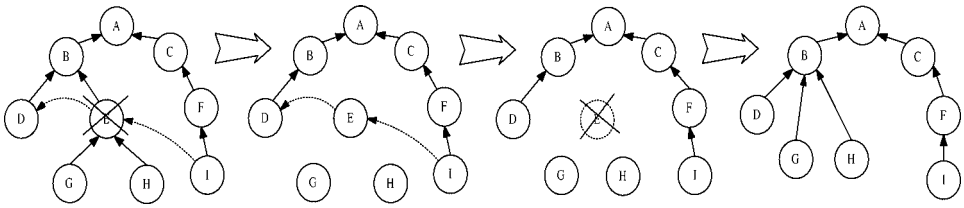

Figure 2. Deleting a catalog community.

Effects:

- Remove the SuperCommunityOf relationship of c (i.e., do `delSub(c,superCatComm(c))`).
- Remove all (direct) SubCommunityOf relationships from c's sub-communities to c (i.e., $\forall c_2$ returned by `subCatComm(c)`, do `delSub(`c_2`,c)`).
- Remove all incoming PeerCommunityOf relationships of c (i.e., $\forall c_3 \in \{c_3|(c_3, c) \in E_2\}$, do `delPeer(`$c_3$`,c)`).
- Remove all outgoing PeerCommunityOf relationships of c (i.e., $\forall c_4 \in \{c_4|(c, c_4) \in E_2\}$, do `delPeer(c,`$c_4$`)`).
- Relink all *orphan* catalog communities to c's super-community (i.e., for every c_2 which has been identified in the second step, do `addSub(superCatComm(c),`c_2`)`).
- Check validity of all bookmarks. $\forall c_5 \in N$, for each bookmark $b \in$ `c`$_5$`.Bookmarks`, if BP(b) contains c, do `delBookmark(`c_2`,b)`.

After deletion of a catalog community E, we delete all bookmarks that reference the community E because such bookmarks are now invalid. That is, any bookmark that contains an action which navigates to, or submit a query to E is identified and removed. Also, the members of E will be automatically registered with `AllCatalog` (i.e., the default catalog community). However, a member can find another catalog community that they wish to join afterwards.

4.2.2. Moving a community. The operation `moveCatComm()` moves a community c from one place to another, by changing its super-community. This operation is used, e.g., when an administrator is convinced that the current super-community of c does not effectively represent the domain of products in c.

For example, in Figure 1, the community `HardDrive` is sub-community of `Peripherals`. Let us assume that observation of user navigation behavior shows that community `Storage` is more suitable super-community for `HardDrive`. This may suggest that it is beneficial to move `HardDrive` to `Storage`. When a community c is moved, all of its sub-communities are moved with c. This creates less overhead, since sub-communities of c do not get affected by the change. The effects of the operation are described in Figure 3 (moving community E from its super-community B to the new super-community C). E's sub-communities, i.e., G and H remain as sub-communities of E. However, since a catalog community cannot be a peer of its super-community, the PeerCommunityOf relationship from H to C has to be deleted.

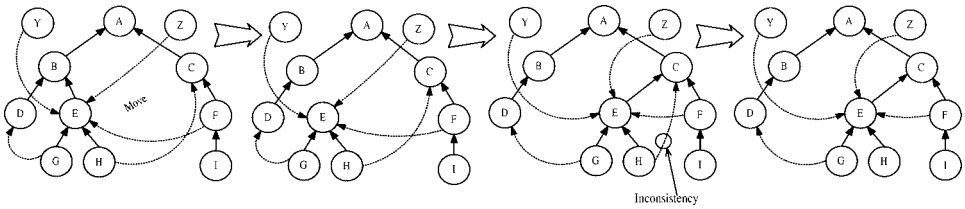

Figure 3. Moving a catalog community.

`moveCatComm(catalogCommunity c, catalogCommunity newSuper)`
Precondition: c, `newSuper` $\in N \wedge$ (`NewSuper`, c) $\notin E_1^+$.
Effects:

- Remove the SuperCommunityOf relationship of c (i.e., do `delSub(c, superCatComm(c))`).
- Add c to sub-community of `newSuper` (i.e., `addSub(c,newSuper)`).
- Remove all outgoing PeerCommunityOf relationships of c's (direct and indirect) sub-communities to `newSuper` (i.e., $\forall c_2 \in \{c_1 | \texttt{indSubCatComm(c)} \wedge (c_1, \texttt{newSuper}) \in E_2\}$, do `delPeer(`$c_2$`,newSuper)`).
- Remove all incoming PeerCommunityOf relationships of `newSuper` to c's (direct and indirect) sub-communities (i.e., $\forall c_2 \in \{c_1 | \texttt{indSubCatComm(c)} \wedge (\texttt{newSuper}, c_1) \in E_2\}$, do `delPeer(newSuper,`$c_2$`)`).
- Remove all PeerCommunityOf relationships between c and `newSuper` (i.e., if (c, `newSuper`) $\in E_2$ do `delPeer(c, newSuper)` or if (`newSuper`, c) $\in E_2$ do `delPeer(newSuper, c)`).
- Check validity of all bookmarks. $\forall c_3 \in N$, for each bookmark $b \in$ `c`$_3$`.Bookmarks`, if BP(b) contains c, do `delBookmark(`c_3`,b)`, where corresponding actions of c in b are either `NavigateToSuper` or `NavigateToSub`.

After moving E, we identify all bookmarks that have become invalid by the operation. Note that even though E is moved, all incoming/outgoing PeerCommunityOf relationships of E are not affected by the operation (i.e., the edges are still valid, pointing to E). Hence, for move operation, we only check for existence of actions that navigate to E through `NavigateToSuper` or `NavigateToSub`.

4.2.3. Merging communities with the same super-community. The operation `mergeCatComm()` merges two communities c and c' which have the same super-community.[7] It is used, for example, when it is observed that the two catalog communities c and c' are always accessed together. Hence, it is beneficial that these two catalog communities are merged, so that the majority of users do not have to visit two separate communities each time. Figure 4 illustrates the effects of `mergeCatComm()`. It shows that a new community is created from merging communities B and C. The super-community of the new community is the super-community of B and C (i.e., A). All sub-communities of B and C (i.e., D, F and G) are sub-communities of the new community. All PeerCommunityOf relationships between B and C, as well as between B and all of C's sub-communities,

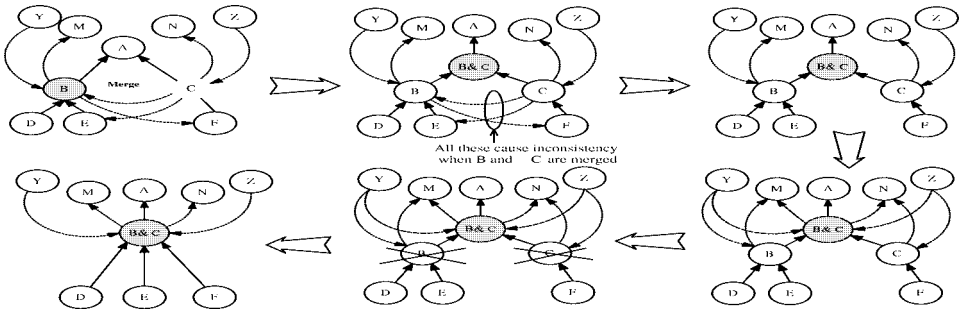

Figure 4. Merging two catalog communities of the same super-community.

C and *B*'s sub-communities should be deleted to maintain the consistency of the eCatalogs-Net. Also, all PeerCommunityOf relationships coming from other communities into *B* and *C* need to be updated, i.e., the PeerCommunityOf relationships would refer to the name of the new community *B&C*, instead of either *B* or *C*.

`mergeCatComm (catalogCommunity c, catalogCommunity c′, String newName)`
Precondition: $c, c' \in N \wedge p = \texttt{superCatComm}(c)$, $p' = \texttt{superCatComm}(c') \wedge p = p'$ (so, let p be the super-community of both c and c').
Effects:

- Create a new community definition which merges the definitions c and c' (i.e., `createCatComm(newName,gi,m,gs)`, where `gi` = `c.GeneralInfo` ∪ `c′.GeneralInfo`, `m` = `c.Members` ∪ `c′.Members` and `gs` = `c.CommProdAttr` ∪ `c′.CommProdAttr`).
- Let c_{new} be the newly merged community. Set temporary Sub(Super)CommunityOf relationships of c_{new}. p becomes c_{new}'s super-community and c, c' are sub-communities of c_{new} (i.e., do `delSub(c,p)`, `delSub(c′,p)`, `addSub(`c_{new}`,p)`, `addSub(c,`c_{new}`)`, `addSub(c′,`c_{new}`)`).
- Remove all PeerCommunityOf relationships between c and c' (i.e., if $(c, c') \in E_2$ do `delPeer(c,c′)` or if $(c', c) \in E_2$ do `delPeer(c′,c)`).
- Remove all PeerCommunityOf relationships between any sub-community of c and any sub-community of c' (i.e., $\forall c_1 \in \{c_1 | \texttt{subCatComm}(c) \wedge (c_1, c') \in E_2\}$, do `delPeer(`$c_1$`,c′)`, $\forall c_2 \in \{c_2 | \texttt{subCatComm}(c') \wedge (c_2, c) \in E_2\}$, do `delPeer(`$c_2$`,c)`).
- Replace references to c (respectively, c') in all PeerCommunityOf relationships of c (respectively, c') with c_{new} (i.e., $\forall c_1 \in \{c_1 \in N | (c_1, c) \in E_2\}$, do `addPeer(`$c_1$`,`$c_{\text{new}}$`)`. $\forall c_2 \in \{c_2 \in N | (c, c_2) \in E_2\}$, do `addPeer(`$c_{\text{new}}$`,`$c_2$`)`).
- Delete c and c' (i.e., do `delCatComm(c)`, `delCatComm(c′)`).
- Check validity of all bookmarks. $\forall c_3 \in N$, for each bookmark $b \in$ `c`$_3$`.Bookmarks`, if BP(b) contains (x, c) or (c, x), where $x \neq c$, do `updateBookmark(`c_3`,c,`c_{new}`,b)`. Also, for (x, c') or (c', x), where $c' \neq x$, do `updateBookmark(`c_3`,c′,`c_{new}`,b)`.

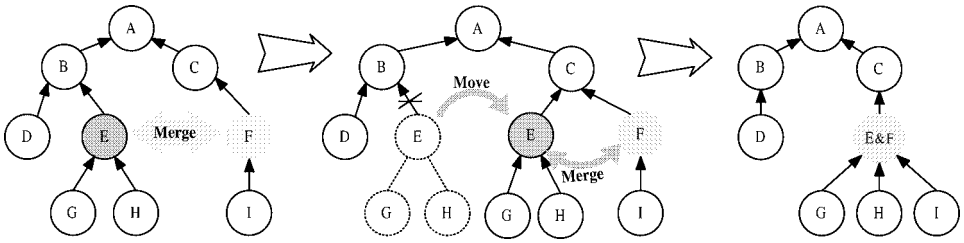

Figure 5. Merging two catalog communities having different super-communities.

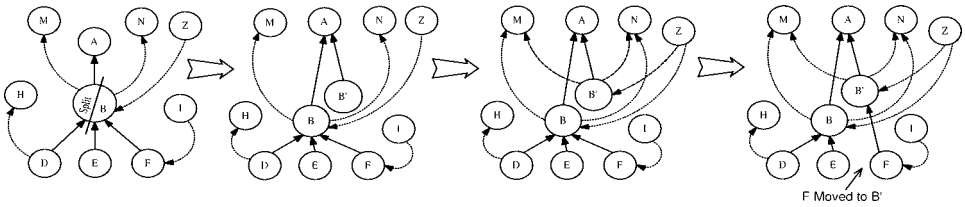

Figure 6. Splitting a catalog community.

As for checking validity of bookmarks, from looking at Figure 4, it is clear that the operation keeps all incoming/outgoing PeerCommunityOf relationships of B and C. The only concern is that both B and C are now replaced by $B\&C$. Therefore, for all communities whose bookmark path contains references to either B or C, we update those references in bookmarks with the new community name.

4.2.4. Merging communities with different super-communities. A situation may exist where it is beneficial to merge two catalog communities, but they have different super-communities. In this case, we first move one of the two communities so that both of them would have the same super-community and then merge them using `mergeCatComm()`. For example, in Figure 5, to merge communities E and F, `moveCatComm(E,C)` is used first to move E to C and then `mergeCatComm(E,F)` is called to merge E and F.[8]

4.2.5. Splitting a community. The operation `splitCatComm()` splits an existing catalog community into two separate communities. This operation is used, e.g., when it is observed that the community represents a domain (described by community attributes) which can be divided into smaller subdomains. This situation is illustrated in Figure 6. Note that as a result of split, one new community is created out of an existing one. The definition of the existing community is updated to reflect this change (e.g., remove community attributes, or members that have been moved to the new community).

Figure 6 illustrates that when the community B is split, a new community B' is created out of B. All incoming and outgoing PeerCommunityOf relationships of B are inherited by B'. Also, if it is necessary, some of the sub-communities of B can be moved to B'.

`splitCatComm(catalogCommunity c, GeneralInfo gic, String Name, GeneralInfo gi, Query mq, CommProdAttr cpa, setOfCommunities sub)`

Precondition: $c \in N \wedge \forall c_i \in N, \ell(c_i) \neq$ `Name`.

Effects: Let $p =$ `superCatComm`(c) and c_{new} the new catalog community.

- Create a new catalog community and initialize its attributes and sub-community relationships using the operation parameters (i.e., do `createCatComm(Name,gi,m,cpa)` where `m` is the result of the query `mq`).
- Add the new SuperCommunityOf relationship of c_{new} (i.e., do `addSub(`c_{new}`,p)`).
- Create new PeerCommunityOf relationships: (i) for every incoming PeerCommunityOf relationship of c (i.e., $\{x \in N | (x, c) \in E_2\}$), do `addPeer(x,`$c_{\text{new}}$`)` (ii) for every outgoing PeerCommunityOf relationship of c (i.e., $\{y | (c, y) \in E_2\}$), do `addPeer(`c_{new}`,y)`. This way, all incoming/outgoing PeerCommunityOf relationships of c are inherited by c_{new}.
- Update the `GeneralInfo` of c with `gic` (this operation mainly affects the domain of c).
- For each community $c_i \in$ `sub`, do `moveCatComm(`c_i`,`c_{new}`)`.
- Check validity of all bookmarks. $\forall c_1 \in N$, for each bookmark $b \in$ `c`$_1$`.Bookmarks`, if BP(b) contains `c`, do `delBookmark(`c_1`,b)`.

Since B has been split into two communities, for any action in a bookmark path that refers to B, it is difficult to automatically decide whether to replace B with B' or leave it as it is. Currently, the definition of operation is set to delete the bookmarks that contain B. However, it is also a possibility that we give an administrator the choice of either deleting it, or updating it manually when the bookmark is identified.

The split of an existing catalog community needs careful consideration about "how" each element (i.e., attribute or relationships) in the community should be treated. It is an administrator's responsibility (e.g., person who is performing the changes) to decide how the attributes `GeneralInfo`, `CommProdAttr`, `Members` and SubCommunityOf relationships should be initialized. This information is specified via operation parameters. We will see in the next section, that the use of interaction patterns will help initializing the attribute `CommProdAttr` of both communities.

Parameters of the `splitCatComm()` include the community being split, specification of its `GeneralInfo`, and specification of elements of the new community. The specification of elements of the new community includes the following: `Name`, `GeneralInfo`, `Query`, `CommProdAttr`, and a set of sub-communities to be moved to the new one. `Query` is a query which will be used by the operation to select members to be moved. Note that sub-communities and members are from the community being split.

5. Predefined interaction sequences

Predefined interaction action sequences represent foreseeable user catalog navigation behavior. In our approach, we use these sequences of actions to help identify situations where

the organization of an eCatalogs-Net may be improved by means of restructuring operations. If a particular sequence of actions has prevalent occurrences, it should be recognized as a recurring user navigation pattern. Each navigation pattern suggests a restructuring operation. This result in continuous improvement of the organization of an eCatalogs-Net based on user interaction patterns. A *Predefined Interaction Sequence* (PIS) is formally defined as follows:

Definition 9 (Predefined Interaction Sequence (PIS)). $\mathcal{A}$ denotes the set of all permissible user actions (see Table 1). A predefined interaction sequence PIS of length n ($n > 0$) is a vector of ordered user actions $\text{PIS} = \langle a_1, a_2, \ldots, a_n \rangle$ where $a_i \in \mathcal{A}$ ($i = 1, \ldots, n$).

For a given PIS, there may exist a session s such that the exact order of actions in PIS can be found in s. A predefined interaction sequence is matched against each session in the processed log file to check whether the sequence exists in the session. We refer to the number of occurrence of a PIS in the log file as *Frequency*. A formal definition and other related issues with calculating the *confidence* of a pattern will be presented in Section 6. In the following subsections, we present a set of predefined interaction sequences. It should be noted that, in addition to the predefined interaction sequences introduced, more patterns can be identified and defined as needed.

5.1. Merging communities

This subsection introduces a generic sequence that describes situations where merging of communities may be beneficial. We also identify some interesting sequences which represent special cases of the generic sequence.[9] It is worth noting that in any of the sequences we introduce, it is important to identify the target catalog community the user was seeking in the particular sequence. Generally, in this paper, we use the action `SubmitQuery` as the factor that defines the target community, since it is the most appropriate action in indicating a user's strong interests in a community. However, an administrator may decide to choose other actions or define new ones such as `PurchaseItem` for the same purpose.

Definition 10 ($\text{PIS}_{\text{GenericMerge}}$). $\text{PIS}_{\text{GenericMerge}}$ which represents the situations where two communities are always queried together, is

$$\text{PIS}_{\text{GenericMerge}} = \langle \texttt{SubmitQuery}(c_i, q_1), a_1, \ldots, a_n, \texttt{SubmitQuery}(c_j, q_2) \rangle,$$

where $c_i, c_j \in N$, $a_k \in \{\texttt{NavigateToSub}, \texttt{NavigateToSuper}, \texttt{NavigateToPeer}\}$ ($k = 1, \ldots, n$), and q_1, q_2 are global query attributes (i.e., community attributes).

$\text{PIS}_{\text{GenericMerge}}$ captures interaction sequences where users, within a catalog community c_i, first submit a query then perform several navigation actions to reach a catalog community c_j from where they finally submit another query (see Figure 7). In the following, we present particular cases of $\text{PIS}_{\text{GenericMerge}}$.

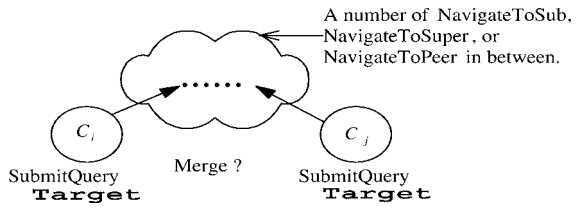

Figure 7. Generic PIS for merging two catalog communities.

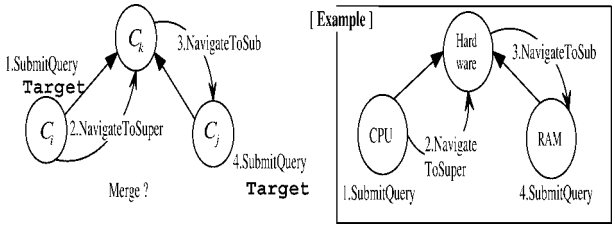

Figure 8. PIS for Merge 1.

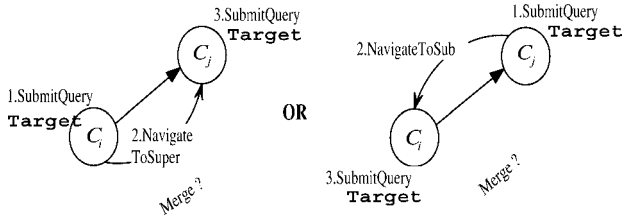

Figure 9. PIS for Merge 2 (two possible sequences).

Merge 1. As shown in Figure 8, $\text{PIS}_{\text{Merge1}}$ is a particular case of $\text{PIS}_{\text{GenericMerge}}$, where two sub-communities of the same super-community are always accessed together without using any PeerCommunityOf relationship. In the example of Figure 8, determining prices of a CPU as well as a RAM requires visiting two catalog communities. If this sequence is observed frequently, it may imply that the two catalog communities should be merged to form one catalog community.

Definition 11 ($\text{PIS}_{\text{Merge1}}$). $\text{PIS}_{\text{Merge1}}$ is described as

$$\begin{aligned}\text{PIS}_{\text{Merge1}} = \langle &\texttt{SubmitQuery}(c_i, q_1), \texttt{NavigateToSuper}(c_i, c_k),\\ &\texttt{NavigateToSub}(c_k, c_j), \texttt{SubmitQuery}(c_j, q_2)\rangle,\end{aligned}$$

where $c_i, c_j, c_k \in N$ and $(c_i, c_k), (c_j, c_k) \in E_1$.

Merge 2. The $\text{PIS}_{\text{Merge2}}$ (see Figure 9) shows a situation where a catalog community and its super-community are always queried together. We have two possible sequences depending on whether the super-community is queried first.

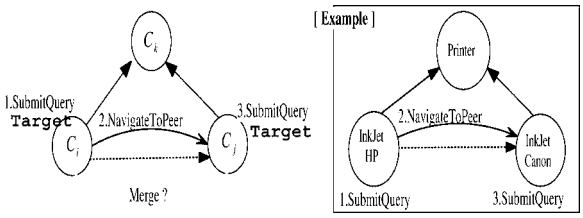

Figure 10. PIS for Merge 3 (a).

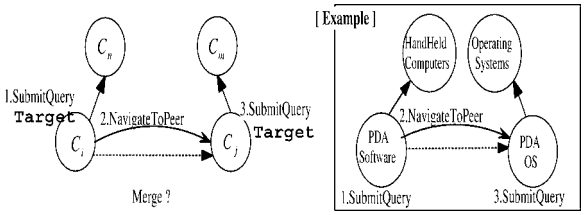

Figure 11. PIS for Merge 3 (b).

Definition 12 ($\text{PIS}_{\text{Merge2}}$). $\text{PIS}_{\text{Merge2}}$ has two possibilities and they are defined as

$$\text{PIS}_{\text{Merge2}} = \langle \texttt{SubmitQuery}(c_i, q_1), a(c_i, c_j), \texttt{SubmitQuery}(c_j, q_2) \rangle,$$

where $a \in \{\texttt{NavigateToSuper}, \texttt{NavigateToSub}\}$, $c_i, c_j \in N$, and $(c_i, c_j) \in E_1 \cup E_1^{-1}$.

Merge 3. $\text{PIS}_{\text{Merge3}}$ shows a similar situation as $\text{PIS}_{\text{Merge1}}$ using `NavigateToPeer` instead of `NavigateToSub` or `NavigateToSuper`. There are two possible sequences (cases (a) and (b) from Figures 10 and 11) depending on whether c_i and c_j have the same super catalog community or not.

Definition 13 ($\text{PIS}_{\text{Merge3}}$). $\text{PIS}_{\text{Merge3}}$ is described as

$$\text{PIS}_{\text{Merge3}} = \langle \texttt{SubmitQuery}(c_i, q_1), \texttt{NavigateToPeer}(c_i, c_j), \texttt{SubmitQuery}(c_j, q_2) \rangle,$$

where $c_i, c_j \in N$ and $(c_i, c_j) \in E_2$.

5.2. *Splitting a community*

A catalog community may be split, if a subset of community attributes are always queried together and the subset can represent a specific domain by itself. One way to detect this situation is to observe the way the community attributes are queried. The following pattern is used to identify a subset of attributes that are always queried together. In this pattern, an administrator has a specific catalog community in mind (c_i) that she/he wants to examine for possibility of splitting, and a set of attributes she/he predicts to be queried together.

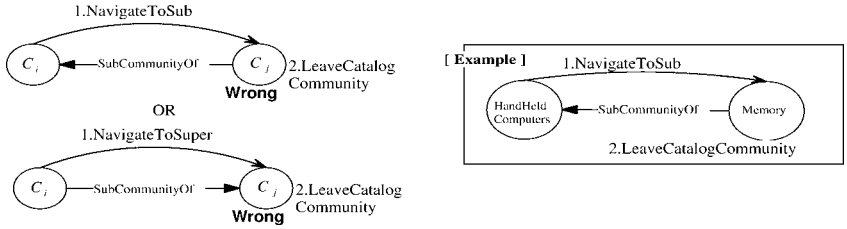

Figure 12. PIS for deleting a catalog community.

Definition 14 ($\text{PIS}_{\text{split}}$). $\text{PIS}_{\text{split}}$ which represents the pattern for splitting a catalog community, is

$$\text{PIS}_{\text{split}} = \langle \texttt{SubmitQuery}(c_i, \text{“}\texttt{attr}_1, \ldots, \texttt{attr}_n\text{”}) \rangle,$$

where $c_i \in N$, $n \leqslant$ number of community product attributes in c_i, and $\texttt{attr}_1, \ldots, \texttt{attr}_n$ are community attributes that are likely to be queried together.

5.3. Deleting a community

If there is a catalog community that users are constantly leaving without performing any further action, it may be beneficial to delete it. This situation is illustrated in Figure 12. If it is observed that the community is obsolete (i.e., has no useful existence), the community can be deleted.

Definition 15 ($\text{PIS}_{\text{DelComm}}$). $\text{PIS}_{\text{DelComm}}$ which represents a situation where `NavigateToSub` or `NavigateToSuper` is always followed by `LeaveCatalogCommunity`, is

$$\text{PIS}_{\text{DelComm}} = \langle a(c_i, c_j), \texttt{LeaveCatalogCommunity}(c_j) \rangle,$$

where $c_i, c_j \in N$, $(c_i, c_j) \in E_1 \cup E_1^{-1}$, and $a \in \{\texttt{NavigateToSub}, \texttt{NavigateToSuper}\}$.

5.4. Moving a community

If users are constantly leaving community c without doing any further action, it may be worthwhile to consider moving it to other location (i.e., to different super-community) where the domain of products is more relevant to the community c. Having the super-community which correctly represents the community's product domain will increase the chance of getting the user access.

In such case, we need to identify where c should be moved to. To find a new super-community for c, we use *Expected Locations* [25]. Given that the goal of a user is to find a catalog community c, the expected locations of c are locations in which the user thought

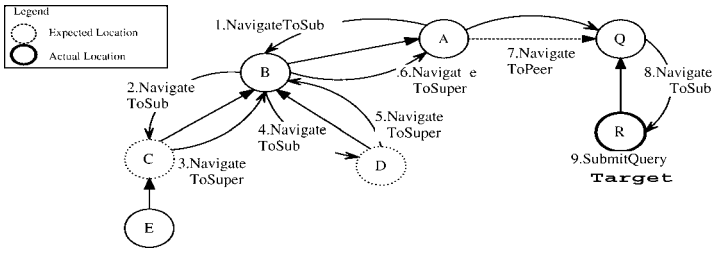

Figure 13. Expected Locations.

(i.e., predicted) to find c, but turns out to be false predictions. That is, c was located somewhere else. Figure 13 explains the concept of expected locations in which the user visits community C and D before finding (and submitting query to) the community R. Therefore, it can be assumed that the user was looking for the community R, and the communities C, and D were the expected locations for it.

In our approach, we identify expected locations of a given community from users' interaction actions. Assume that a user is looking for a community c. While navigating eCatalogs-Net, when the user backtracks from any community c' without doing further action, we assume the community c' is an expected location of c. Those backtrack actions are marked by interaction patterns such as a sequence of ⟨`NavigateToSub(c1, c2)`, `NavigateToSuper(c2, c1)`⟩ [10] or ⟨`NavigateToPeer(c1,c2)`, `NavigateToPeerc(c2,c1)`⟩. There may be more than one expected location for a given community (e.g., R had two expected locations, C, D in Figure 13). The following pattern is used to identify all expected locations of a community c. Among those expected locations, we choose the one with most prevalent occurrences.

Definition 16 (PIS_{move}). PIS_{move} represents a situation where a number of backtrack actions occurred before a SubmitQuery action:

$$\text{PIS}_{\text{move}} = \langle a_1, a_2, \ldots, a_n, \texttt{SubmitQuery}(c, q)\rangle,$$

where c is the community which is considered for move, and $\langle c_0, c_1, c_2, \ldots, c_n\rangle$ (corresponding sequence of communities of $\langle a_1, a_2, \ldots, a_n\rangle$) satisfies:

- $c_n = c$, $c_{i-1} \neq c_i$, and $(c_{i-1}, c_i) \in E_1 \cup E_1^{-1} \cup E_2$ $(i = 1, \ldots, n)$,
- one of the c_i $(i = 1, \ldots, n-1)$ is the expected location.

5.5. *Updating a PeerCommunityOf relationship*

We now discuss predefined interaction sequences designed to observe the usage patterns of an existing PeerCommunityOf relationship and identify situations where we can upgrade/downgrade its weight (via `updatePeer()` operation). The predefined interaction sequences of this category will help: (i) decide whether to keep or delete the PeerCommunityOf relationship, (ii) determine which PeerCommunityOf relationships are more

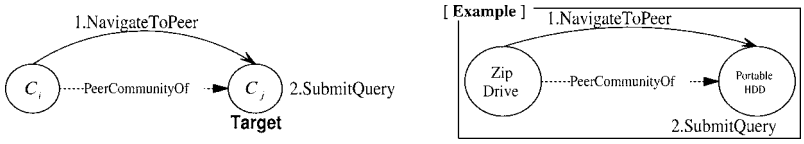

Figure 14. PIS for upgrading the weight of PeerCommunityOf relationship.

relevant for a given community, and (iii) detect situations where the creation of a new PeerCommunityOf relationship is beneficial (i.e., result in improved organization of an eCatalogs-Net).

5.5.1. Upgrading the weight of a PeerCommunityOf relationship. This PIS concerns upgrading the weight of a given PeerCommunityOf relationship in order to consolidate the relevancy of this relationship. Figure 14 describes a situation where users navigate from community c_i, via PeerCommunityOf relationship, to community c_j, and submit a query to c_j. This indicates that c_j is the target community users are looking for. The PeerCommunityOf relationship in c_i positively contributed in finding the target. The related PIS is defined as follows:

Definition 17 ($\text{PIS}_{\text{upgrade}}$). $\text{PIS}_{\text{upgrade}}$ which represents the situation where `NavigateToPeer` is always followed by `SubmitQuery`, is

$$\text{PIS}_{\text{upgrade}} = \langle \texttt{NavigateToPeer}(c_i, c_j), \texttt{SubmitQuery}(c_j, q) \rangle,$$

where $c_i, c_j \in N$, $(c_i, c_j) \in E_2$, and q is global query attributes (i.e., community attributes).

When there exist sequences in the log file that give this pattern a high frequency (subject to the administrator's interpretation), the weight of PeerCommunityOf relationship from c_i to c_j is increased. If the weight of a PeerCommunityOf relationship in a source community c_i reaches the higher threshold (e.g., value of 0.95 out of 1), the target of the relationship (i.e., c_j) is considered to be highly relevant alternative to c_i.

5.5.2. Downgrading the weight of a PeerCommunityOf relationship. PISs of Figure 15 show interaction sequences where users follow a PeerCommunityOf relationship and arrive at a community c_j. However, they ultimately leave the community without performing any further action. This may indicate that c_j is not relevant to these users. Figure 15 displays two different possibilities of leaving a community. Users can either perform `LeaveCatalogCommunity` or `NavigateToPeer`. The former will lead users to default community *AllCatalog* and the latter is only possible when there exist an inverse PeerCommunityOf relationship (i.e., from c_j to c_j).

Definition 18 ($\text{PIS}_{\text{downByLeave}}$). $\text{PIS}_{\text{downByLeave}}$ which represents a situation where `NavigateToPeer` is followed by `LeaveCatalogCommunity`, is

$$\text{PIS}_{\text{downByLeave}} = \langle \texttt{NavigateToPeer}(c_i, c_j), \texttt{LeaveCatalogCommunity}(c_j) \rangle,$$

where $c_i, c_j \in N$, $(c_i, c_j) \in E_2$.

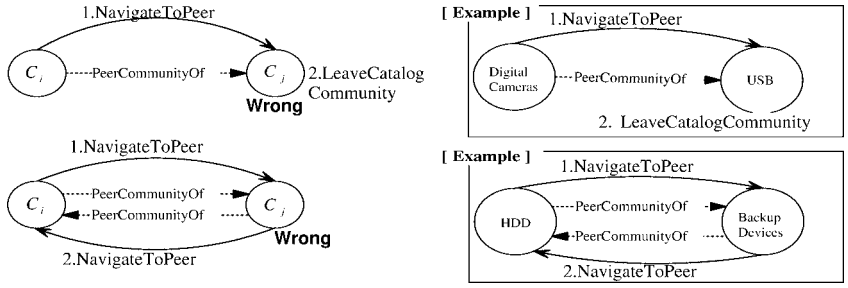

Figure 15. Two PISs for downgrading the weight of a PeerCommunityOf relationship.

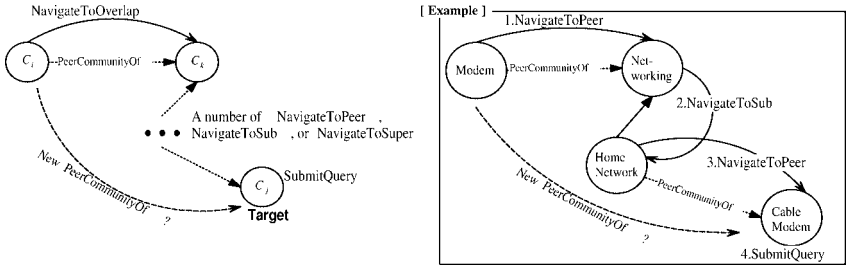

Figure 16. PIS for creating a PeerCommunityOf relationship.

Definition 19 ($\text{PIS}_{\text{downByPeer}}$). $\text{PIS}_{\text{downByPeer}}$ which represents a situation where `NavigateToPeer` is followed by another `NavigateToPeer` to go back to the source community of the former `NavigateToPeer`, is

$$\text{PIS}_{\text{downByPeer}} = \langle \texttt{NavigateToPeer}(c_i, c_j), \texttt{NavigateToPeer}(c_j, c_i) \rangle,$$

where $c_i, c_j \in N$, $(c_i, c_j), (c_j, c_i) \in E_2$.

Whenever this pattern (i.e., one of the two described sequences) gets a high frequency, the weight of the PeerCommunityOf relationship from c_i to c_j is decreased.

5.5.3. Creating a PeerCommunityOf relationship. Here, we introduce another PIS that may suggest the creation of a new PeerCommunityOf relationship (via `addPeer()` operation). This PIS is used to identify communities that are constantly used as stop-overs. Therefore, it may be beneficial to create direct PeerCommunityOf relationships so that users can bypass them (see Figure 16, for example).

Definition 20 ($\text{PIS}_{\text{createPeer}}$). $\text{PIS}_{\text{createPeer}}$ which represents a situation where there are one or more navigational actions between `NavigateToPeer` and `SubmitQuery`, is

$$\text{PIS}_{\text{createPeer}} = \langle \texttt{NavigateToPeer}(c_i, c_k), a_1, \ldots, a_n, \texttt{SubmitQuery}(c_j, q) \rangle,$$

where $a_p \in \{\texttt{NavigateToSub}, \texttt{NavigateToSuper}, \texttt{NavigateToPeer}\}$ $(p = 1, \ldots, n)$, $c_i, c_j, c_k \in N$, $(c_i, c_k) \in E_2$, and $(c_i, c_j) \notin E_1 \cup E_1^{-1} \cup E_2$.

If the PeerCommunityOf relationship between c_i and c_j exists already, the weight of the relationship will be increased.

5.5.4. Deleting a PeerCommunityOf relationship. For the purpose of deleting an existing PeerCommunityOf relationship, we consider patterns defined for downgrading the weight of a PeerCommunityOf relationship (i.e., $\text{PIS}_{\text{downByLeave}}$, $\text{PIS}_{\text{downByPeer}}$). When it is observed that the weight of a PeerCommunityOf relationship in a community reaches the lower threshold (e.g., value of 0.05 out of 1), the relationship is considered to irrelevant. Hence, it can be removed.

5.6. Creating a bookmark

There is no predefined pattern specifically designed to create a new bookmark. However, an administrator creates new bookmarks according to his/her observation over user's navigation patterns. An administrator can define a sequence of actions and submit it as a query. For example, if an administrator suspects that from `Hardware`, there are many users who navigate to `HDD` and then to `PortableHardDrives` to query for prices, she/he can query for the following sequence of actions as a pattern:

```
⟨ NavigateToSub(Hardware,HDD),
   NavigateToSub(HDD,PortableHardDrives),
   SubmitQuery(PortableHardDrives,q)⟩.
```

If the pattern is found to appear frequently, these particular actions can be recorded as a bookmark and associated with the community `Hardware` for the convenience of users.

6. Relevance of patterns

In this section, we provide two definitions, namely, *frequency* and *relevance*. They are used to decide whether a PIS can be considered as a pattern for which a restructuring operation is suggested.

Definition 21 (Frequency). A frequency of a predefined interaction sequence PIS, denoted by Freq(PIS), is defined as

$$\text{Freq(PIS)} = \text{number of occurrences of PIS in the processed log file.}$$

The frequency of a predefined interaction sequence is used to decide whether the result of the match is significant enough to consider performing eCatalogs-Net restructuring operations. We discuss some of the issues that arise from using the patterns.

Table 3. Conflicting and consolidating patterns

	$PIS_{upgrade}$	$PIS_{downByLeave}$	$PIS_{downByPeer}$	$PIS_{createPeer}$	$PIS_{DelComm}$	PIS_{Merge1}	PIS_{Merge2}	PIS_{Merge3}	PIS_{split}	PIS_{move}
$PIS_{upgrade}$	·	−	−	+	n	n	n	+	n	n
$PIS_{downByLeave}$	·	·	+	−	+	n	n	−	n	n
$PIS_{downByPeer}$	·	·	·	−	+	n	n	−	n	n
$PIS_{createPeer}$	·	·	·	·	−	n	n	n	n	n
$PIS_{DelComm}$	·	·	·	·	·	−	−	−	−	−
PIS_{Merge1}	·	·	·	·	·	·	n	n	−	n
PIS_{Merge2}	·	·	·	·	·	·	·	n	−	n
PIS_{Merge3}	·	·	·	·	·	·	·	·	−	n
PIS_{split}	·	·	·	·	·	·	·	·	·	n
PIS_{move}	·	·	·	·	·	·	·	·	·	·

Legend: n = no conflict, − = conflict, + = consolidation.

First, there is an issue of conflicting patterns where one pattern suggests a certain restructuring operation, whereas another pattern leads to a different operation on the same relationships or communities. For instance, it is possible that the pattern $PIS_{upgrade}$ shows that the weight of PeerCommunityOf relationship between community *A* and *B* needs to be upgraded, but at the same time, the pattern $PIS_{downByPeer}$ may suggest that the same relationship should be downgraded.

Second, there is an issue of knowing patterns that can consolidate each other. We refer to these patterns as consolidating patterns. These patterns, when used together, can reinforce each other's findings. For example, suppose that the pattern $PIS_{downByLeave}$ suggests that PeerCommunityOf relationship between community *A* and *B* should be downgraded. When $PIS_{downByPeer}$ pattern also suggests downgrading of the same relationship, it confirms the need for a restructuring operation. Table 3 lists the identified conflicting and consolidating patterns among the predefined interaction patterns presented in this paper. In the following, we discuss a quantitative measure of relevance. Relevance of a PIS, noted Relevance(PIS), is defined as follows:

$$\text{Relevance(PIS)} = \frac{\text{RF(cons)}}{\text{RF(conf)}} \cdot \log \frac{\text{RF(cons)}}{\text{RF(conf)}}, \tag{1}$$

where RF(cons) is a relevance factor of all consolidating patterns of PIS and is equal to

$$\text{RF(cons)} = \frac{\text{Freq(PIS)} + A}{\text{Freq(PIS)}}, \tag{2}$$

where *A* is the sum of frequency of all consolidating patterns of PIS. RF(conf) is a relevance factor of all conflicting patterns of PIS and is equal to

$$\text{RF(conf)} = \frac{\text{Freq(PIS)} + B}{\text{Freq(PIS)}}, \tag{3}$$

Table 4. Examples of choosing δ

RF(cons)/RF(conf)	RF(cons)/RF(conf) · log RF(cons)/RF(conf)
1	0
1.5	0.26413689
2	0.60205999
2.5	0.99485002
3	1.43136376

where B is the sum of frequency of all conflicting patterns of PIS. By using (2) and (3), (1) becomes

$$\text{Relevance(PIS)} = \frac{\text{Freq(PIS)} + A}{\text{Freq(PIS)} + B} \cdot \log \frac{\text{Freq(PIS)} + A}{\text{Freq(PIS)} + B}. \tag{4}$$

In (4), $(\text{Freq(PIS)} + A)/(\text{Freq(PIS)} + B)$ shows the degree of deviation of the consolidating patterns from the conflicting ones. This ratio indicates different relationship between $\text{Freq(PIS)} + A$ and $\text{Freq(PIS)} + B$.

Case 1. If $(\text{Freq(PIS)} + A)/(\text{Freq(PIS)} + B)$ is close to 1, that is $A \simeq B$, the frequency of all conflicting patterns is almost equal to the frequency of all conflicting patterns.

Case 2. If $(\text{Freq(PIS)} + A)/(\text{Freq(PIS)} + B)$ is less than 1, that is $A < B$, then conflicting patterns occur more frequentlyfrequently than consolidating ones.

Case 3. If $(\text{Freq(PIS)} + A)/(\text{Freq(PIS)} + B)$ is greater than 1, that is $A > B$, then consolidating patterns occur more frequently than conflicting patterns.

In the context of finding relevant patterns, it is obvious that we are only interested in Cases 1 and 3. Therefore, the logarithm function is taken since $\log(\text{Freq(PIS)} + A)/(\text{Freq(PIS)} + B)$ is less than 0 for Case 2. The weight $\text{Freq(PIS)} + A$ is added to the measure to take into account how frequently the consolidating patterns occur. Another weight $1/(\text{Freq(PIS)} + B)$ is introduced to indicate how valid the correlation between consolidating and conflicting patterns is. Thus, the higher the value Relevance(PIS), the more relevant a pattern PIS. With the above definition of the relevance factors of consolidating and conflicting patterns, we arrive at the following definition about the relevance of patterns.

Definition 22 (Relevance of a pattern). A pattern PIS is relevant if $\text{Relevance(PIS)} \geqslant \delta$, where $\delta(\geqslant 0)$ is a threshold given by a system administrator.

Table 4 shows examples of how an administrator chooses the value of the threshold δ. For instance, if the administrator decides to say that the pattern is relevant if the sum of its consolidating patterns is at least twice as much the sum of conflicting patterns, the value of δ will be set to 0.6.

7. Implementing WebCatalog$^{\text{Pers}}$

In this section, we present the WebCatalog$^{\text{Pers}}$ prototype which implements the concepts and functionalities of the eCatalogs-Net and its adaptive communities. The purpose of this

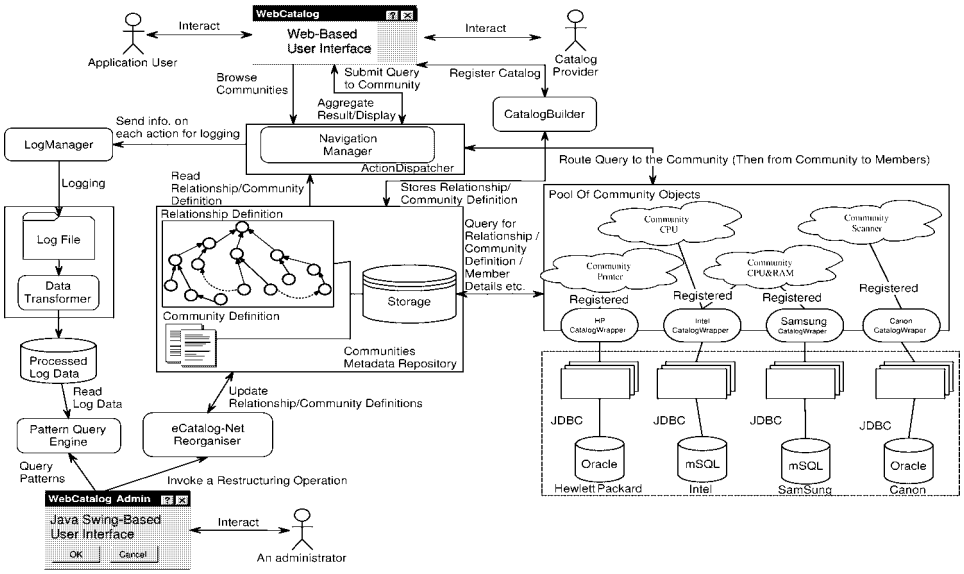

Figure 17. Overview of WebCatalog$^{\text{Pers}}$ architecture.

implementation is to validate the feasibility of main concepts of eCatalogs-Net, predefined interaction sequences, and restructuring operations. So far, the implementation has shown that the ideas of WebCatalog$^{\text{Pers}}$ are consistent with one another and are realizable using existing technologies.

Overall, WebCatalog$^{\text{Pers}}$ have been implemented using Java. For persistent storage (log data and metadata repository), Oracle 8i database is used. The metadata repository in WebCatalog$^{\text{Pers}}$ stores community definitions and the relationships. The interaction with the database is done via JDBC. The architecture of WebCatalog$^{\text{Pers}}$ is shown in Figure 17. In the following, we present the main components of WebCatalog$^{\text{Pers}}$.

7.1. `CatalogBuilder`

`CatalogBuilder` allows community creators to build a community and also provides product catalog providers with methods to register their local product catalog to a community. To cater for community creation and catalog registration, `CatalogBuilder` has two classes: `CommunityWrapper` and `CatalogWrapper`.

The `CommunityWrapper` class has a set of attributes which describes the underlying product catalogs (i.e., community attributes). One of the methods provided by the class is called `register_catalog` which takes information from a local product catalog provider and register the catalog to the community as a member. When a catalog provider registers with a community, the participating catalog provider is represented in the community as an instance of `CatalogWrapper` class. Another method `query_catalog`

in CommunityWrapper class is used to send a user's query (e.g., global query) to corresponding member of the community (i.e., an instance of CatalogWrapper).

The CatalogWrapper class has a set of attributes which describes a product catalog (i.e., local product catalog attributes). One of the methods in the class is called trans_query which loads a translator program that translates (i.e., maps) a global query. This is expressed using community attributes to a local query which is expressed using local product catalog attributes. The output of the query is translated back into the global format and made accessible through the method get_query_result.

7.2. NavigationManager

The Web-based user interface of WebCatalog^{Pers} provides navigation methods for customers to interact with eCatalogs-Net. NavigationManager provides two classes, namely, ECatalog and ECatalogNet. The class ECatalogNet provides methods to manipulate eCatalogs-Net graph such as addSub(), getPeerCommunity(). The ECatalog class provides methods to manipulate single node representing a community (e.g., setCommName(), getMembers()). Four JSP programs, namely, ViewCatalog, ShowMembers, QueryHandler and ErrorPage use the classes to display the eCatalogs-Net and handle user actions (e.g., navigation, querying, listing of members etc.). Especially, ActionDispatcher servlet, which acts as a dispatcher of all user's actions, diverts each user action to a proper submodule. For example, all navigation actions (NavigateToSub, NavigateToSuper, NavigateToPeer) are delivered to ViewCatalog.

The Web-based user interface also allows catalog providers to register their catalogs with communities. It uses the register_catalog method of CommunityWrapper and the registration details provided by the catalog providers (e.g., product attributes, product domain, attribute mapping information etc.).

7.3. LogManager

The ActionDispatcher servlet also sends parameters of each action to the LogManager for logging. The LogManager provides a Java class called UserActionLogger. This class contains methods which creates log entries. Logged data is fed into the DataTransformer which performs sessionizing, extraction of global query attributes. An administrator can operate this module periodically (e.g., every week).

7.4. Reorganizer

We implemented a Java Swing-based GUI application to be used by an administrator. It interacts with Pattern Query Engine and eCatalogNetReorganiser. The Pattern Query Engine processes queries for the predefined interaction sequences. Currently, we have implemented a set of pattern query using PL/SQL, which is written

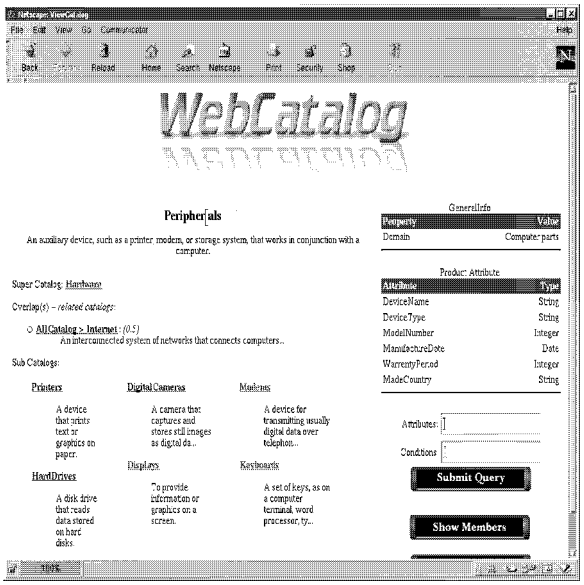

Figure 18. The main user interface – for navigation.

for each pattern we have introduced in this paper. It searches, session by session, for any sequence that matches the predefined interaction sequence and returns the actual matching sequences with a frequency. `eCatalogNetReorganiser` mainly communicates with the repository. An administrator can invoke any of the defined restructuring operations through this interface, which involves manipulation of the relationships definitions and community definitions.

7.5. Computer and related services application

In order to demonstrate the viability of this architecture, we took examples of product catalogs from the domain of *computer and related services* and created 27 communities (as in Figure 1), each community having 3–4 members (i.e., catalog providers). The `Catalog-Wrapper` class of product catalog is associated with a Java application which accesses the product catalog's local database and handles the translation between global and local query. The application supports creation of a community, registration of a product catalog, querying a product catalog through a community. Figures 18 and 19 illustrate a main usage scenario of an end user. Figure 18 depicts the main user interface where users can start navigating through the catalog communities. Relationships between communities are presented as hyperlinks so that users can easily move between communities by clicking links. The interface displays *current* (i.e., the community a user is currently looking at) community's `GeneralInfo` and `Product Attributes`. Users can submit a global query to the community using the product attributes, which will be passed on to all of the community's members. Figure 19 shows a screen shot of the system, after a user has se-

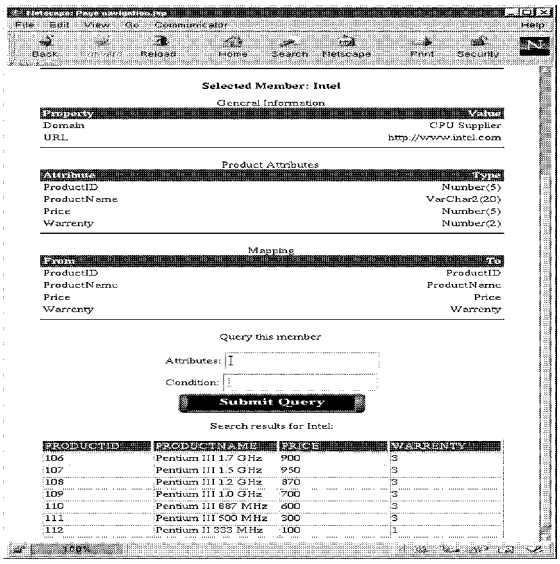

Figure 19. Querying a member catalog.

lected a member of a community and submitted a query to the member. The result of the query is displayed on the same screen.

8. Evaluation

In our initial studies, we have concentrated on the particular question of "does restructuring increase the chance of user finding the target?". Having said that, one would argue that any user could find a target if she/he was given enough time to search. It should be noted that our initial experiments are conducted under simulated scenarios, in that we restricted the users' search in terms of number of moves (e.g., mouse clicks) they can make. Hence, it could be said that primary goal of this experiment is to demonstrate that given the same constraint (i.e., limited number of moves), more users find targets after restructuring operations.

There are other ways to measure the improvement such as comparing the total number of communities that users had to visit or total time taken (as in seconds/minutes) to find targets. We plan to incorporate all these measures in the imminent future.

8.1. Evaluation environment

As mentioned under implementing, the eCatalogs-Net (see Figure 1) used in the experiments represents an integrated view of 27 catalog communities in *computer and related services*. A key element in our experiment is a log file obtained from the users' (i.e., customers) searching/querying behavior. We used *task agents* that played the role of

Table 5. A likelihood table

Community name	Likelihood
AllCatalog	2
Hardware	7
Software	5
Internet	8
Components	4
Peripherals	6
...	...
Modem	7
CableModem	10
...	...

customers who wanted to find out information about the products. A Java class called `AgentFactory` was used to create agents. More precisely, the class `AgentFactory` implements a software component made up of a container and a pool of objects which represent agents. The container is a process that runs continuously, listening to a socket, through which an instantiation message from a predefined script (used to create an agent) is received. Once an agent is created, it interacts with the class `ECatalogNet` which provides various methods for exploring the community relationships (e.g., `getSubCommunityOf()`, `getPeerCommunityOf()` etc.).

The agent's search and query behavior is based on the same search and query strategy which is presented in Section 2.3. The agents are equipped with two kinds of information for autonomous interaction with communities. First, the agents has access to the relationships (i.e., Sub, PeerCommunityOf) between communities. The other information provided to the agents is called *Likelihood Table*. An agent is given a name of the community to find (called the target community). In the likelihood table, given the target community, every community in eCatalogs-Net is assigned a number value which represents a degree of "closeness" (i.e., relevance) of the community to the target community. Hence, the higher the value, the *more likely* the community will lead the agent to the target. We will refer to this value as a *likelihood* and the list of likelihood values as a *likelihood table*. An example of a likelihood table is shown in Table 5. The likelihood values in the example table are produced given that the target community is `CableModem`.

However, fixing the likelihood values in the table creates a predictable agents' interaction sequence. Agents should be able to make spontaneous and irregular decisions, resulting in unpredictable behavior. Therefore, we introduced a variant factor which would diverge a likelihood value. Every time, an agent is given the likelihood values, the agent recalculates all likelihood values according to this factor before starting navigation. Table 6 shows an example of such likelihood table. In the example, given original likelihood values, new values are created with the variant factor (VF) of "$\pm$10%."

The agent takes the input parameters in Table 7 to run. For the purpose that stated earlier in Section 8, we limited the `MaxMove` (refer to Table 7) to 14 for all experiments.

Also, a few parameters are defined in relation to the likelihood tables. Descriptions of the likelihood table related parameters used in the experiments and their value ranges are shown in Table 8.

Table 6. Likelihood recalculated by a variant factor

Community name	Recalculated (±10%) likelihood
AllCatalog	2.2
Hardware	7.7
Software	5.2
Internet	7.5
Components	4.4
Peripherals	5.8
...	...
Modem	7.5
CableModem	10
...	...

Table 7. Agent input parameters

Parameter	Description
`RLoc`	Location of the repository of eCatalogs-Net.
`LogName`	Name of the log file that the agent's actions will be logged out to.
`LHT`	Name of the file that contains likelihood table(s).
`Target`	Name of the target community to find.
`Query`	Query to submit when the target is found.
`MaxMove`	Number of moves an agent can make before it gives up searching.

Table 8. Experimental parameters

Parameter	Description and setting
NT	Number of likelihood tables used. Each table contains different values. Setting: 1 (single table), 2 (two tables), and 3 (three tables).
RV	Range of likelihood values for likelihood tables. Setting: 1 to 10, 1 to 50, and 1 to 100.
VF	Value of the variant factor. Setting: 5%, 10%, and 15%.

NT represents the number of likelihood tables used in an experiment. As for the parameter NT, we produced three different versions of likelihood tables. One of the three tables is created by us as domain experts. Naturally it contains reasonable and precise values for the target. We refer to this table as the "expert table." Another table is created by non-experts. We simulated, by asking four people who are not familiar with the domain to create the likelihood tables, and then averaged the four values. This table contains less precise values than the first. This table is referred to as the "non-expert table." The last table contains unreasonable, least precise likelihood values and is called a "random table."

RV denotes a range of likelihood values in a likelihood table. It has three different settings, 1–10, 1–50 and 1–100. The higher the range, the more precisely the community relationships can be described by the likelihood values. VF represents the value of the variant factor for a likelihood table. It also has three settings, 5%, 10% and 15%. The higher the variant factor, the bigger the deviation from given likelihood values.

8.2. Experiments and results

We now describe the results of experiments that investigated the effect of two restructuring operations; `addPeer()` and `moveCatComm()` with the parameters shown in Table 8. The experiments carried out were based on two pre-established simulation scenarios.

8.2.1. First scenario. In the first scenario, we experimented on a PeerCommunityOf relationship. 3000 agents were created and given the task of finding the community `CableModem`. The result from initial experiments showed that the following navigation pattern (which is an instance of $\text{PIS}_{\text{createPeer}}$) got a frequency of 590:

```
NavigateToPeer(Modem,Networking),
   NavigateToSub(Networking,HomeNetwork),
   NavigateToSub(HomeNetworking,CableModem),
   SubmitQuery(CableModem,query)
```

This constituted about 28% of the agents who found the target. As $\text{PIS}_{\text{createPeer}}$ suggests, we created a new PeerCommunityOf relationship from `Modems` to `CableModem`. That is, the eCatalogs-Net now had a new edge from `Modems` to `CableModem`. Then we ran the agents again to see the "before and after" effects of the restructuring operation. Experiment settings and results are discussed in the following.

Experiment 1: Varying NT (RV = 1–100 and VF = 15%). In this experiment, we measured the improvement made and the effect of having different numbers of likelihood tables. The values of RV and VF are fixed. Each likelihood table created represented a group of people with a different level of understanding. In NT = 1 setting (i.e., a single likelihood table), the "expert table" is used, NT = 2 setting uses "expert table" + "non-expert table." NT = 3 uses "expert table" + "non-expert table" + "random table."

As can be seen in Figure 20, there was consistent improvement in the number of agent-found targets after the operation with different settings of NT. This means that restructuring operations can benefit all groups of people with different understandings of the community relationships. Another interesting observation is that the biggest improvement is made on NT = 3, i.e., with most diverse group of people in terms of the level of understanding. However, the graph also shows that the number of agent-found targets decreased as NT increased. This indicates that the group of users with a better understanding of the domain is more likely to find the target easily.

Experiment 2: Varying RV (NT = 1 and VF = 15%). This experiment measured the improvement made, and the effect of having a different range of likelihood values. RV was set at 1 to 100, 1–50 and 1–10. In Figure 21, again we saw consistent improvement consistently over different settings. Overall, we seemed to get the bigger improvement (biggest gap between before and after) when RV was set at 1–100. When RV was 1–50, the number of agent-found targets was the highest. There was only a little improvement (compared to other settings) when RV was set at 1–10, and also, at this setting the least number of agent-found targets. We believe this is because the 1 to 10 range probably was too small to precisely described the relationships between communities.

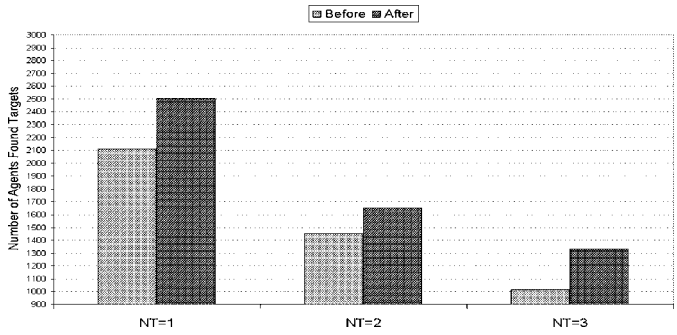

Figure 20. Varying NT (RV = 1–100, VF = 15%): first scenario.

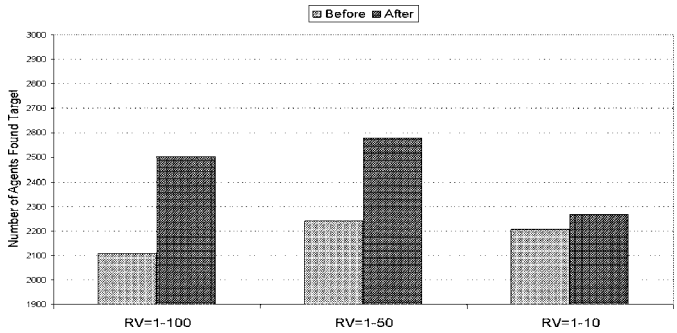

Figure 21. Varying RV (NT = 1, VF = 15%): first scenario.

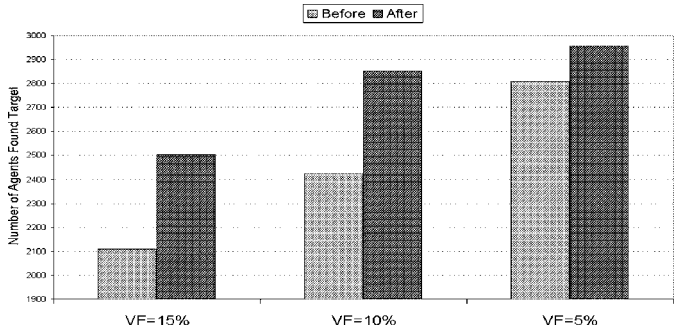

Figure 22. Varying VF (NT = 1, RV = 1–100%): first scenario.

Experiment 3: Varying VF (NT = 1 and RV = 1–100). In this experiment, we measured the improvement made by the operation and the effect of different values of the variant factor. In the experiments, VF varied from 15% to 10%, and then to 5%. As shown in Figure 22, irrespective of creation of the relationship, as VF decreases, the more agents were able to find targets. The variant factor (VF) random-

izes the likelihood values from a given table. This result demonstrates that agents are likely to find the target if their likelihood values are less deviated from the given likelihood values. Given the fact that the NT = 1 uses the most precise values for the target, this result can be interpreted as indicating that the user whose understanding does not deviate much from that of an expert is more likely to find targets easily. Also, the biggest improvement of "before and after" was made when VF was 15 (i.e., highest deviation). This indicates that the restructuring of eCatalogs-Net can be of most benefit to the user when her/his understanding deviates considerably from the expert.

8.2.2. Second scenario. In the second scenario, we experimented on moving a community to a new super-catalog community. In the initial structure of eCatalogs-Net (Figure 1), `HardDrive` is sub-catalog community of `Peripherals`. The likelihood tables built by both domain experts, and non-experts expected that `HardDrive` would be under `Storage` (i.e., `Storage` scored higher likelihood value than `Peripherals`). In the initial runs, 3000 agents were created and given the task of finding the community `HardDrive` and submitting a query "`Select BrandName, Capacity, Price From HardDrive`." For the second runs, we moved `HardDrive` from `Peripherals` to `Storage` and ran the agents again.

Experiment 1: Varying NT (RV = 1–100 and VF = 15%). This experiment in the second scenario (NT = 1, 2 and 3), we saw a huge improvement when a single likelihood table ("expert table") is used. The other two settings also showed impressive improvements, but it was clearly not as huge as the NT = 1 setting. The expert table expected the target `HardDrive` to be under `Storage`. The fact that we moved the `HardDrive` to the experts' expected location improved the chances of finding the target.

Experiment 2: Varying RV (NT = 1 and VF = 15%). The result of this experiment in the second scenario repeated what we have found in the first scenario (Figure 24). That is, when RV was set at 1 to 100, 1–50 and 1–10, we saw constant improvement made by the restructuring operation over the three settings. Also, the biggest improvement was seen when RV = 1–100. When RV was set at 1–10, regardless of the move, we recorded the least number of agent-found targets.

Experiment 3: Varying VF (NT = 1 and RV = 1–100). The second scenario of this experiment also confirmed what we found in the first scenario. As shown in Figure 25, dramatic improvements were made after the operation. However, irrespective of the move, as VF decreases, the more agents were able to find targets. Also, the biggest improvement was made when VF was 15 (i.e., highest deviation).

Overall, across all experiment settings, we saw consistent improvements after restructuring eCatalog-Net. This demonstrates that adaptive restructuring of e-catalogs can help users have more streamlined, easier navigation/search experiences.

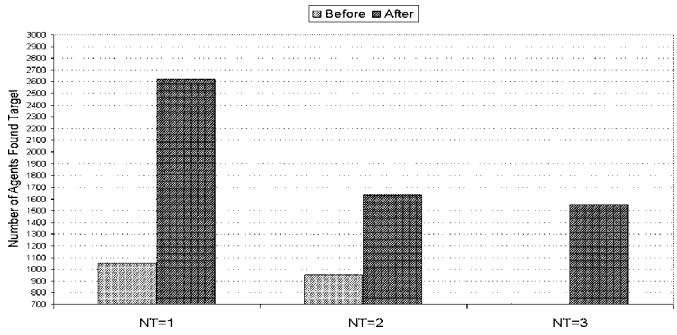

Figure 23. Varying NT (RV = 1–100, VF = 15%): second scenario.

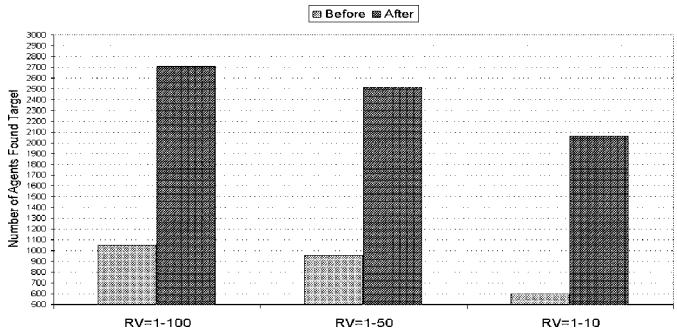

Figure 24. Varying RV (NT = 1, VF = 15%): second scenario.

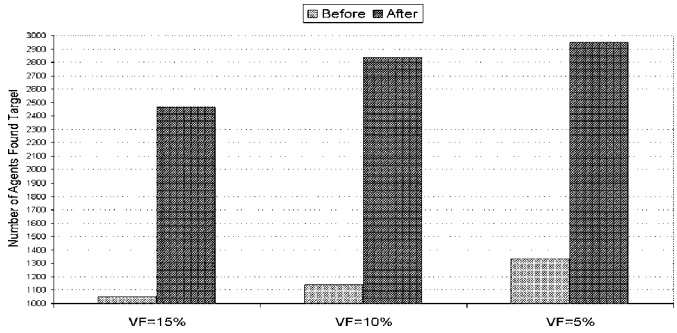

Figure 25. Varying VF (NT = 1, RV = 1–100%): second scenario.

9. Related work

We discuss two major related areas; namely, adaptive Web sites, and Web usage mining. We use the term "adaptive" in the context of changing Web site organization to help users have better interaction experience while using the site. Perkowitz and Etzioni [21,22] au-

tomatically construct index pages that supplement an existing organization by looking at co-occurring pages, so that users can easily locate pages that are conceptually and strictly related to one topic. The construction of supplementary index pages is a non-destructive approach because the original Web site remains unchanged. However, the cost of building/managing such index pages separately, especially in dynamic communities environment, can be quite expensive. In [18], a technique that discovers the gap between Web site designer's expectation and user's behavior is proposed. The technique uses inter page conceptual relevance versus inter page access co-occurrence. Srikant and Yang [25] developed an algorithm to identify "expected locations" of a Web page and create a link from the expected location to the page. The basic idea is that if a target page of a user's single visit is known, the target page's expected locations can be identified by keeping track of user's backtrack points. The correctness of what the algorithm produces heavily depends on the ability to identify the target page of a single visit which is not a straightforward task. Toyoda and Kitsuregawa [26] use existing hyperlink structure analysis algorithms to build a Web community chart, which users can navigate from one community to other related communities. They extract related pages from a given page (a seed), then investigate the relationships between two Web pages based on how each page drives other pages as a related page.

It is worth noting that, while basic principles of this area are complementary to our work, most approaches only deal with Web pages, which is quite different from the concept of communities we proposed. For example, in [26], a Web community is defined as "*a collection of Web pages created by an individual or an association that has a common interest on a specific topic*." In our work, communities are individual and autonomous entities (rather than a network of Web pages) with which users and members of the community can have various interactions (submitting a query to, invoking operations from, register with, etc.).

Our work is also related to mining access patterns from Web server logs (e.g., Web usage mining and navigation sequence mining of data mining area) [1,14,15] in that we use a log that records sequences of user actions as a basis for our reasoning.

Datta et al. [10] use a Web usage mining concept to dynamically predict user's next behavior and to make a recommendation. In [5], Hypertext Probabilistic Grammar is also used to predict the user's navigation path. The authors of [9,16] discuss issues and processes involved in preparation/transformation of data from Web server logs to a format suited for purposes of mining. Sadri et al. [23] proposed an extension of SQL to allow fast querying over sequential patterns in a Web server log file. In typical sequence of Web usage mining, an access pattern is a sequence of visited Web documents which have a large occurrence frequency. It extracts frequently visited nodes, or nodes that are visited together, but this kind of access pattern does not reflect how users navigate the imposed structure. A Web usage mining (WUM) system to evaluate effectiveness of the Web site organization is proposed in [4,24]. It uses a concept of g-sequence to model sequences of navigation of users. We use a similar concept to model sequence of user interaction actions with communities.

Another work worth mentioning is [30], in which decision trees are used to automatically construct catalogs based on popularity of product items (i.e., frequency of visits) and weighted product attributes. The algorithm of construction is designed in a way that the

depth of product hierarchy (which is a tree) is minimized, pushing the popular product items/attributes to upper levels so that customers can find them easily (with fewer clicks). However, it does not discuss the ongoing adaptivity of the catalogs. The salient features of our work are:

- We specifically model permissible user interaction actions to track user navigation behavior. Hence, the log file is not a series of Web pages (which cannot give meaning to the access of pages), rather it is a series of user interaction actions. Using user actions, which carry explicit meaning, enables us to infer more precise semantics behind user navigation patterns ([10] addresses modeling of user actions in the context of predicting the user's next move in browsing Web documents).
- Although it is possible to feed our log file data to data mining algorithm to *discover existing navigation patterns*, we use quite a different approach, in which we have *predefined sequences of catalog interaction actions*, and query those sequences to confirm the existence of some user navigation patterns. These patterns are used as heuristics for improving the organization of catalogs by means of catalog restructuring operations.
- We used the concept of user navigation mining in a novel application domain, dynamic reorganization of online product catalog communities. We proposed operations for restructuring communities also as means to support making a decision to perform such operations, which has not been attempted before to the best of our knowledge.

10. Conclusion and future work

In this paper, we proposed a usage-centric approach for transforming and improving integrated catalog structure and organization. We illustrated, through simulated experiments, the viability of the proposed approach and demonstrated that restructuring operations increase the chances of the user finding his/her targets.

Ongoing work includes case studies to assess improvement of catalog organization as results of restructuring operations in a realistic environment. Also, we are working towards a new approach in which the relationships between communities (SubCommunityOf and PeerCommunityOf) are discovered automatically, rather than assigned manually, by comparing product attributes and descriptions [12,17]. There are also plans to extend the proposed idea to handle the case where people with similar navigation patterns are considered [16,27].

Acknowledgements

The work of the second author, Dr. Boualem Benatallah, is partially supported by the ARC (Australian Research Council) discovery grant DP0211207. We are specially grateful to Dr. Fabio Casati from HP Laboratories for his valuable comments and suggestions on the earlier version of this article.

Notes

1. It should be noted that the focus of this paper is not on catalog integration, but rather on adapting organization of integrated catalogs based on customer interaction patterns.
2. It is equivalent to say that `Peripherals` is a super-community of `Printer`. We assume that, each catalog community has at most one super-community, hence this relationship is automatically implied.
3. Srikant and Yang [25] uses a similar strategy for browsing and searching Web documents.
4. A target community of an action a_i is the destination of a_i. For example, a target community of action `NavigateToSub(CableModem)` is `CableModem`.
5. UID and TS are not shown for clarity reasons.
6. The symbol $\bullet$ represents the ordered sequence of actions in a bookmark.
7. Note that, in this paper, we only consider merging of two communities, but the operation can be generalized to more than two communities.
8. It is also possible that the catalog community F is moved towards catalog community E and merged under B.
9. Note that, even though actions in Table 1 do not include source catalog community parameter, we add them when defining PIS for clarity reasons.
10. The sequence could be $\langle$`NavigateToSuper(c1, c2)`, `NavigateToSub(c2, c1)`$\rangle$, depending on where the user is.

References

[1] C. C. Aggarwal and P. S. Yu, "Data mining techniques for personalization," *Bulletin of the Technical Committee on Data Engineering* 23(1), March 2000.
[2] Ariba Inc., `http://www.ariba.com`
[3] B. Benatallah, M. Dumas, Q. Z. Sheng, and A. H. H. Ngu, "Declarative composition and peer-to-peer provisioning of dynamic Web services," in *Proceedings of the International Conference on Data Engineering*, San Jose, USA, February 2002.
[4] B. Berendt and M. Spiliopoulou, "Analysing navigation behaviour in Web sites integrating multiple information systems," *VLDB Journal*, Special Issue on Databases and the Web 9(1), 2000, 56–75.
[5] J. Borges and M. Levene, "Data mining of user navigation patterns," in *Proceedings of the Workshop on Web Usage Analysis and User Profiling (WEBKDD'99)*, San Diego, CA, August 1999.
[6] A. Bouguettaya, B. Benatallah, and A. K. Elmagarmid, *Interconnecting Heterogeneous Information Systems*, Kluwer Academic, Boston, 1998.
[7] A. Bouguettaya, B. Benatallah, L. Hendra, M. Ouzzani, and J. Beard, "Supporting dynamic interactions among Web-based information sources," *IEEE Transactions on Knowledge and Data Engineering* 12(5), September/October 2000, 779–801.
[8] commerceOne Inc., `http://www.commerceone.com`
[9] R. Cooley, B. Mobasher, and J. Srivastava, "Data preparation for mining World Wide Web browsing patterns," *Journal of Knowledge and Information Systems* 1(1), 1999.
[10] A. Datta, K. Dutta, D. E. VanderMeer, K. Ramamritham, and S. B. Navathe, "An architecture to support scalable online personalization on the Web," *VLDB Journal* 10(1), 2001, 104–117.
[11] Dublin Core Metadata Initiative (DCMI), `http://www.dublincore.org`
[12] D. Fensel, Y. Ding, and B. Omelayenko, "Product data integration in B2B e-commerce," *IEEE Intelligent Systems* 16(4), July/August 2001.
[13] J. Jung, D. Kim, S. Lee, C. Wu, and K. Kim, "EE-Cat: Extended electronic catalog for dynamic and flexible electronic commerce," in *Proceedings of the IRMA2000 International Conference*, Anchorage, AK, IDEA Group Publishing, May 2000.
[14] J. A. Konstan, B. N. Miller, and D. Maltz, "GroupLens: Applying collaborative filtering to Usenet news," *Communications of the ACM* 40(3), March 1997.
[15] D. Mladenic, "Text-learning and related intelligent agents: A survey," *IEEE Intelligent Systems* 14(4), July/August 1999, 44–54.

[16] B. Mobasher, R. Cooley, and J. Srivastava, "Automatic personalization based on Web usage mining," *Communications of the ACM* 43(8), August 2000.
[17] G. Modica, A. Gal, and H. M. Jamil, "The use of machine-generated ontologies in dynamic information seeking," in *Proceedings of Sixth International Conference on Cooperative Information Systems (CoopIS 2001)*, Trento, Italy, 2001.
[18] T. Nakayma, H. Kato, and Y. Yamane, "Discovering the gap between Web site designers' expectations and user's behaviour," in *Proceedings of 9th International World Wide Web Conference*, Amsterdam, May 2000.
[19] S. Navathe, H. Thomas, M. Satits, and A. Datta, "A model to support e-catalog integration," in *Proceedings of the IFIP Conference on Database Semantics*, Hong Kong, Kluwer Academic, April 2001.
[20] H. Paik, B. Benatallah, and R. Hamadi, "Usage-centric adaptation of dynamic e-catalogs," in *Proceedings of 14th International Conference on Advanced Information Systems Engineering*, Toronto, Canada, May 2002.
[21] M. Perkowitz and O. Etzioni, "Adaptive Web sites," *Communications of the ACM* 43(8), August 2000.
[22] M. Perkowitz and O. Etzioni, "Towards adaptive Web sites: Conceptual framework and case study," *Artificial Intelligence* 118, 2000, 245–275.
[23] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi, "A sequential pattern query language for supporing instant data mining for e-services," in *Proceedings of the 27th VLDB Conference*, Roma, Italy, September 2001.
[24] M. Spilopoulou, "Web usage mining for Web site evaluation," *Communications of the ACM* 43(8), August 2000.
[25] R. Srikant and Y. Yang, "Mining Web logs to improve website organization," in *Proceedings of 10th International World Wide Web Conference*, Hong Kong, May 2001.
[26] M. Toyoda and M. Kitsuregawa, "A Web community chart for navigating related communities," in *Proceedings of 10th International WWW Conference*, Hong Kong, May 2001.
[27] K. Wu, C. C. Aggarwal, and P. S. Yu, "Personalization with dynamic profiler," Technical Report, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY, 2001.
[28] XML Common Business Library, `http://www.xcbl.org`
[29] G. Yan, W. Ng, and E. Lim, "Product schema integration for electronic commerce – A synonym comparison approach," *IEEE Transactions on Knowledge and Data Engineering* 14(3), May/June 2002.
[30] D. Yang, W. Sung, S. Yiu, D. Cheung, and W. Ho, "Construction of online catalog topologies using decision trees," in *Proceedings of Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000)*, Milpitas, CA, 8–9 June 2000.