Introduction to Logical Information Systems

Sébastien Ferré, Olivier Ridoux

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Introduction to Logical Information Systems

Sébastien Ferré, Olivier Ridoux

N˚4540

Septembre 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4540--FR+ENG

INRIA
RENNES

Introduction to Logical Information Systems

Sébastien Ferré, Olivier Ridoux

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n ° 4540 — Septembre 2002 — 25 pages

Abstract: Logical Information Systems (LIS) use logic in a uniform way to describe their contents, to query it, to navigate through it, to analyze it, and to maintain it. They can be given an abstract specification that does not depend on the choice of a particular logic, and concrete instances can be obtained by instantiating this specification with a particular logic. In fact, a logic plays in a LIS the role of a schema in data-bases. We present the principles of logical information systems, the constraints they impose on the expression of logics, and hints for their effective implementation.

Key-words: information systems, information search and retrieval, query formulation, representation languages, deduction and theorem proving.

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Introduction aux systèmes d'information logiques

Résumé : Les Systèmes d'Information Logiques (SIL) utilisent la logique de façon uniforme pour décrire leur contenu, pour l'interroger, pour y naviguer, pour l'analyser et pour le maintenir. On peut leur donner une spécification abstraite qui ne dépend pas du choix d'une logique particulière, et des instances concrètes peuvent être obtenues par instanciation de cette spécification à une logique particulière. En fait, une logique joue dans un SIL le rôle d'un schéma dans les bases de données. Nous présentons les principes des systèmes d'information logiques, les contraintes qu'ils imposent sur l'expression des logiques et des éléments pour leur implémentation effective.

Mots-clé : systèmes d'information, recherche d'information, formulation de requêtes, langages de représentation, déduction et démonstration de théorèmes.

1 Introduction

Several researchers have recognized the *name problem* in information systems (Gifford, Jouvelot, Sheldon, and O'Toole, 1991; Gopal and Manber, 1999). In these systems, *things* are given *names* and very often a thing has only a few names (frequently only one). For instance, in the most rudimentary information systems, like hierarchical file systems, a thing is a file and its name is the only path that leads from the root to the file.

The problem with having only a few names is that they must be very carefully chosen to tackle for all future usages of the things. Experience shows that this is impossible to do for a wide range of future usages. For instance, there is no hierarchical organization of many pieces of software that fits all the needs of software: programming, testing, documenting, debugging, etc. A non-IT example is a cook-book. Cook-books are often organized following the course of a meal, and it is thus very difficult to search for a recipe according to other criteria, like the (un)availability of a oven, or a diet constraint. Note that even electronic cook-books often follow this structure. Electronic or not, the classical solution to this problem is to search in the whole information system/document to find the desired thing. However, this is only a partial solution because query language used in search engines are often too restricted.

There have been several attempts for solving the name problem. We mention here only two of them that we have selected because they illustrate the difficulty of the enterprise, SFS (Gifford, Jouvelot, Sheldon, and O'Toole, 1991) and HAC (Gopal and Manber, 1999). They have in common to combine *hierarchical naming* and *boolean querying*.

Hierarchical naming is frequently found in computer tools: e.g., file systems, book-marks, or menus. In this model, searching is done by *navigating* in a classification structure that is often built and maintained manually. Navigating implies a notion of *place*; *being in* a place, and *going to* another place. A notion of *neighborhood* helps specifying the "other places" relatively to the place one is currently in. Many applications require that a place is a place to read from as well as a place to write on.

Boolean querying is often found in information servers such as search engines on the Web (e.g., Google). In this model, searching is done by using *queries*, generally expressed in a kind of propositional logic. A well-recognized difficulty of this model is the necessity of having a good knowledge of the terminology used in the information system, and of having a precise idea of what is searched for. However, it is easier to recognize an object than to describe it, and the necessity of expressing a query can repel casual users.

Then, which search model should be preferred: navigation or querying? In fact, it depends on situations, and it is sometimes needed to use both of them in the same search. For instance, someone could wish to begin his search by a query and then refine it with navigation. Hybrid systems combining hierarchical classification and boolean querying have been proposed in the domain of file systems:
— SFS (*Semantic File System*, Gifford, Jouvelot, Sheldon, and O'Toole, 1991) extends the hierarchical model of usual file systems with *virtual directories* that correspond to queries. These queries concern file properties that are automatically extracted by *transducers*, and are expressed with valued attributes. So, two organization and storage methods coexist: the standard hierarchy that gives a name to files, and virtual directories that enable associative searches on intrinsic file properties. Unfortunately, these two methods cannot be combined freely in general. In particular, virtual directories are not places to write into.
— HAC (*Hierarchy And Content*, Gopal and Manber, 1999) also uses queries to build directories based on file contents, but these directories are integrated in the hierarchy. This enables to combine hierarchy and contents in searches. However, users are allowed to move a file in a directory even if it does not satisfy the query associated to the directory, which results in consistency problems.

The drawback of these hybrid systems is their lack of consistency. Indeed, they have two search models that are not tightly connected, which makes it difficult to switch from one model to the other, and to combine both in the same search.

We propose a scheme called Logical Information Systems (LIS) in which queries are really places to read from and to write into. The scheme is flexible in the sense that the neighborhood relation is very dense (i.e., things have many names). It incurs no inconsistency or dangling links problem, because the neighborhood relation is managed automatically. Finally, it supports both querying and navigation, and arbitrary combinations of both because names and queries belong to the same language (Godin, Missaoui, and April, 1993; Lindig, 1995). This scheme is based on a variant of Formal Concept Analysis (Wille, 1982; Ganter and Wille, 1999) called Logical Concept Analysis (Ferré and Ridoux, 2000).

The article is organized as follows. Section 2 presents the principles of Logical Concept Analysis, and then Section 3 shows how it can be used for navigating and querying. How to create and update the content of a LIS is presented in Section 4. These sections refer to a logic passed as a parameter which is to be used for naming, querying and navigating. Section 5 presents other functions of a LIS like automated updating, data-mining and

learning. Section 6 explains how all this can be done practically. Finally, conclusions and perspectives are given in Section 7.

2 Logical Concept Analysis

The origin of this work is the search for flexible organizations for managing, updating, querying, or navigating in data. In this context, several roles are played by possibly different people: e.g., designer, administrator, and end-user. Hierarchical organizations are not flexible, and updating, querying and navigation are difficult to conciliate (see for instance the view update problem in data-bases (Keller, 1985)). The literature shows that *Formal Concept Analysis* (*FCA*, Ganter and Wille, 1999) is a good candidate for supporting querying and navigation.

The basis of FCA is a *formal context* that associates attributes to objects; objects are the *things* in the introduction, and the collection of attributes associated to one thing/object is its *name*. FCA has received attention for its application in many domains such as in software engineering (Snelting, 1998; Lindig, 1995; Krone and Snelting, 1994). The interest of FCA as a navigation tool in general has also been recognized (Godin, Missaoui, and April, 1993; Lindig, 1995; Vogt and Wille, 1994). However, we feel it is not flexible enough as far as the naming of things is concerned, and the literature on FCA insists more on analyzing a given context than on managing evolving contexts. In this section, we present an extension to FCA that allows for a richer name language.

The variety of application domains bring the need for more sophisticated formal contexts than the mere presence/absence of attributes. For instance, many application domains use numerical values (e.g., lengths, prices, ages), and the need to express negation and disjunction is often felt. In a much more specialized scope, it is imaginable to use the type of software components as search keys (Di Cosmo, 1995). Several enrichments to the attribute structure have been proposed: e.g., many valued attributes (Ganter and Wille, 1999), and first-order terms (Chaudron and Maille, 1998). However, not a single extended FCA framework covers all the concrete domains, and no one can pretend covering all the concrete domains to come.

So, we propose to construct a more general framework for concept analysis, Logical Concept Analysis (LCA), in which the logic of attributes becomes a parameter. This will allow for instantiating the general framework by merely filling in a dedicated logic.

2.1 Logic

Logical Concept Analysis (LCA) is a generalization of FCA, where sets of attributes are replaced by formulas of an (almost) arbitrary logic. More details about the relation with previous works in FCA can be found in (Ferré and Ridoux, 2000), and all proofs can be found in (Ferré and Ridoux, 1999).

We first define what we call a *logic* in this article.

Definition 1 (logic) *A* logic *is a 6-tuple* $(\mathcal{L}, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$*, where*

- $\mathcal{L}$ *is the language of* formulas*,*
- $\sqsubseteq$ *is the* subsumption *relation (pre-order over* $\mathcal{L}$*),*
- $\sqcap$ *and* $\sqcup$ *are respectively* conjunction *and* disjunction *(binary operations),*
- $\top$ *and* $\bot$ *are respectively* tautology *and* contradiction *(constant formulas).*

Such a logic must form a lattice (Davey and Priestley, 1990), whose order is derived in the usual way from the pre-order $\sqsubseteq$*, and such that* $\sqcap$ *and* $\sqcup$ *are respectively the* infimum *(greatest lower bound) and the* supremum *(least upper bound), and* $\top$ *and* $\bot$ *are respectively the* top *and the* bottom*. The notation* $\mathcal{L}$ *can be used as a name for the logic lattice.*

If $f \sqsubseteq g$ and $g \sqsubseteq f$, f and g are called logically *equivalent*, which is denoted as $f \equiv g$; we will consider them as different representations of the same equivalence class, and in fact we will consider that elements of $\mathcal{L}$ are the equivalence classes. We just assume, for practical reasons, that operations $\sqsubseteq, \sqcap$, and $\sqcup$ are computable. Some *semantics* is usually used to define a logic, but we delay the discussion about it until Section 6.2 as we do not need it here. In order to clarify things, here is an example of a logic.

Example 1 (Propositional logic) *An example of logic that can be used in LCA is* propositional logic. *On the syntactic side, the set of propositions $\mathcal{P}$ contains atomic propositions (taken in a set $\mathcal{A}$), formulas* 0 *and* 1, *and is closed under binary connectives $\wedge$ and $\vee$ and unary connective $\neg$. We say that a proposition p is subsumed by another one q if $\neg p \vee q$ is a valid proposition ($p \vDash q$). Then, $(\mathcal{P}, \vDash, \wedge, \vee, 1, 0)$ satisfies Definition 1, because it is the well-known boolean algebra.*

This example shows that though the interface of the logic is limited to the tuple $(\mathcal{L}, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$, an actual logic may have more connectives: e.g., $\neg$ in this example.

We now define the main notions and results of LCA: *context*, *concept lattice*, and *contextualized subsumption*.

2.2 Logical context and Galois connection

A *logical context* plays the role of tables in a database, as it gathers the knowledge one has about objects of interest (e.g., files, bibliographic references, recipes).

Definition 2 (logical context) *A* logical context *(context for short) is a triple $K = (\mathcal{O}, \mathcal{L}, d)$ where:*

- *$\mathcal{O}$ is a finite set of objects,*
- *$\mathcal{L}$ is a logic (as in Definition 1),*
- *d is a mapping from $\mathcal{O}$ to $\mathcal{L}$ that* describes *each object by a formula.*

Then, we define two mappings between sets of objects ($2^{\mathcal{O}}$) and formulas ($\mathcal{L}$) in a context K, that we prove to be a Galois connection (Davey and Priestley, 1990). A first mapping τ_K connects each formula f to its *instances*, i.e., objects whose description is subsumed by f; $\tau_K(f)$ is the *extent* of f. A second mapping σ_K connects each set of objects $O \subseteq \mathcal{O}$ to the most precise formula subsuming all descriptions of objects in O; $\sigma_K(O)$ is the *intent* of O.

Definition 3 (mappings τ_K and σ_K) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context, $O \subseteq \mathcal{O}$, and $f \in \mathcal{L}$,*

- *σ_K (σ for short)* $: 2^{\mathcal{O}} \to \mathcal{L}$, $\sigma_K(O) := \bigsqcup_{o \in O} d(o)$
- *τ_K (τ for short)* $: \mathcal{L} \to 2^{\mathcal{O}}$, $\tau_K(f) := \{o \in \mathcal{O} \mid d(o) \sqsubseteq f\}$

Lemma 1 (Galois connection) *Let K be a context. The pair (σ_K, τ_K) is a* Galois connection *because* $\forall O \subseteq \mathcal{O}, f \in \mathcal{L} : \sigma_K(O) \sqsubseteq f \iff O \subseteq \tau_K(f)$.

In the following, we will drop the subscript K when possible.

Example 2 ($Triv$) *An example context will illustrate the rest of our development on LCA. Context K_{Triv} is deliberately small and simple as it is aimed at illustrating theoretical notions, and not at showing a realistic application of LCA. The logic used in this context is the propositional logic $\mathcal{P}$ (see Example 1) with a set of atomic propositions $\mathcal{A} = \{a, b, c\}$. We define context K_{Triv} by $(\mathcal{O}_{Triv}, \mathcal{P}, d_{Triv})$, where $\mathcal{O}_{Triv} = \{x, y, z\}$, and where $d_{Triv} = \{x \mapsto a, y \mapsto b, z \mapsto c \wedge (a \vee b)\}$.*

A Logical Information System (LIS) is essentially a logical context equipped with navigation and management tools. Formulas serve as queries, and extensions as answers via mapping τ_K. This is only a rough description. We will see in Section 3 that a LIS answer can also be a formula. For illustration purpose, we consider a bibliographical information system.

Example 3 (Bib) *Let $Bib = (\mathcal{O}, \mathcal{P}_v, d)$ be a logical context where objects are bibliographical references, whose description are composed of a type (e.g., article, in-proceedings), a list of authors, a title, and a year of publication. The logic $\mathcal{P}_v$, used for expressing descriptions, is similar to the propositional logic $\mathcal{P}$ (see Example 1) except that atoms are replaced by valued attributes in the form* attr value*: attr is the name of an attribute (e.g.,* `author`, `year`*), and* value *expresses a logical property about the value of the attribute. For instance, numerical attributes can be described in an interval logic (e.g.,* `year in 1990..2000`, `year in 1995`*), and string attributes can be described by a string (e.g.,* `title is "Logical Information Systems"`*) or a substring (e.g.,* `title contains "System"`*). We now show as an example the logical description of our article on LCA (Ferré and Ridoux, 2000).*

```
   type is "InProceedings"
∧ author is "Sébastien Ferré, Olivier Ridoux"
∧ title is "A Logical Generalization of Formal Concept Analysis"
∧ year in 2000
```

In the sequel of this article, the context Bib refers to all ICCS publications until the year 1999, which consists in 209 objects.

2.3 Concept lattice and labelling

A *formal concept*, central notion of LCA, is the association of a set of objects and of a formula, which is stable for the Galois connection (σ, τ).

Definition 4 (formal concept) *In a context $K = (\mathcal{O}, \mathcal{L}, d)$, a* formal concept *(concept for short) is a pair $c = (O, f)$ where $O \subseteq \mathcal{O}$, and $f \in \mathcal{L}$, such that $\sigma_K(O) \equiv f$ and $\tau_K(f) = O$.*

The set of objects O is the concept extent *(written $ext(c)$), whereas formula f is its* intent *(written $int(c)$).*

We write $=^c$ for concept equality. The set of all concepts that can be built in a context K is denoted by $\mathcal{C}_K$, and is partially ordered by $\leq^c$ defined below. The fundamental theorem of LCA is that $\langle \mathcal{C}_K; \leq^c \rangle$ forms a *lattice*, which is finite, hence complete.

Definition 5 (partial order $\leq^c$) *Let c_1 and c_2 be in $\mathcal{C}_K$,*
$c_1 \leq^c c_2 \iff ext(c_1) \subseteq ext(c_2) \quad (\iff int(c_1) \sqsubseteq int(c_2))$.

Theorem 1 (concept lattice) *Let K be a context. The partially ordered set $\langle \mathcal{C}_K; \leq^c \rangle$ is a finite lattice, whose supremum and infimum are as follows for every set of indices J:*

- $\bigvee^c_{j \in J}(O_j, f_j) =^c (\tau_K(\sigma_K(\bigcup_{j \in J} O_j)), \sqcup_{j \in J} f_j)$
- $\bigwedge^c_{j \in J}(O_j, f_j) =^c (\bigcap_{j \in J} O_j, \sigma_K(\tau_K(\sqcap_{j \in J} f_j)))$

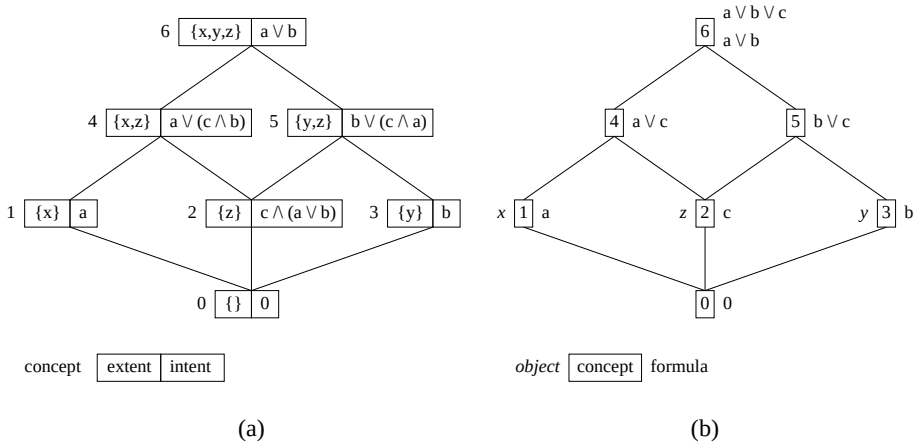


Figure 1: The concept lattice of context K_{Triv} (a) and its labelling (b).

Example 4 ($Triv$) *Figure 1.(a) represents the Hasse diagram of the concept lattice of context K_{Triv} (introduced in Example 2). Concepts are represented by a number and a box containing their extent on the left, and their intent on the right. The higher concepts are placed in the diagram the greater they are for partial order $\leq^c$. It can be observed that the concept lattice is not isomorphic to the power-set lattice of objects $\langle 2^{\mathcal{O}}; \subseteq \rangle$. Indeed, set $\{x, y\}$ is not the extent of any concept, because $\tau(\sigma(\{x, y\})) = \tau(a \vee b) = \{x, y, z\}$.*

To make the concept lattice more readable, it is possible to label it with formulas and objects. Mapping μ labels with a formula f the concept whose extent is the extent of f. Mapping γ labels with an object o the concept whose intent is the intent of o, that is its description.

Definition 6 (labelling) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context, $o \in \mathcal{O}$, and $f \in \mathcal{L}$,*

- μ_K *(μ for short)* $: \mathcal{L} \to \mathcal{C}_K$, $ext(\mu_K(f)) = \tau_K(f)$
- γ_K *(γ for short)* $: \mathcal{O} \to \mathcal{C}_K$, $int(\gamma_K(o)) \equiv d(o)$.

The interesting thing with this labelling is that it enables to retrieve all data of the context: an object o satisfies ($\sqsubseteq$) a formula f in some context K if and only if the concept labelled with o is below ($\leq^c$) the concept labelled with f in the concept lattice of K.

Lemma 2 (labelling) *Under the conditions of Definition 6,*

$$d(o) \sqsubseteq f \iff \gamma_K(o) \leq^c \mu_K(f) \ .$$

Example 5 ($Triv$) *Figure 1.(b) represents the same concept lattice as Figure 1.(a) (see also Example 4), but its concepts are decorated with the μ labelling instead of with the extents and intents. Formulas of the form $\bigvee A$ where $A \subseteq \mathcal{A}$ ($\bigvee \emptyset \equiv 0$) are placed on the right of the concept that they label. For instance, concept 1 is labelled by formula a (i.e., $\mu(a) =^c 1$). In Figure 1.(b) we have restricted labels to be formulas of the form $\bigvee A$, but it is only to have a finite number of labels that are not all in the formal context.*

2.4 Contextualized subsumption

In most contexts, it is possible to order some properties, although they are not comparable by $\sqsubseteq$. For instance, if in some context every bird flies, then we can say that property "bird" is *contextually subsumed* by the property "fly", although we have not necessarily $bird \sqsubseteq fly$ in $\mathcal{L}$. We introduce a *contextualized subsumption* as a generalization of implications between attributes, that are used in standard CA for knowledge acquisition processes (Ganter and Wille, 1999; Snelting, 1998).

Definition 7 (contextualized subsumption) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context, and $f, g \in \mathcal{L}$. One says that f is* contextually subsumed *by g in context K, which is noted $f \sqsubseteq_K g$, if and only if $\tau_K(f) \subseteq \tau_K(g)$, i.e., if every object that satisfies f also satisfies g.*

Every arc of $\sqsubseteq_K$ is called a contextualized implication.

Contextualized subsumption has a close connection with the concept lattice.

Theorem 2 (contextualized subsumption vs. concept lattice) *Let K be a context. The partially ordered set of formulas $\langle \mathcal{L}; \sqsubseteq_K \rangle$, derived in the usual way from the pre-order $\sqsubseteq_K$, is isomorphic to the concept lattice $\langle \mathcal{C}_K; \leq^c \rangle$. The morphism from formulas to concepts is μ_K (Definition 6); and the morphism from concepts to formulas is int (Definition 4).*

A context plays the role of a theory extending the subsumption relation and enabling new entailments. Contextualized subsumption can also be seen as a means for extracting knowledge from contexts. Thus, two kinds of knowledge can be extracted: knowledge about context by deduction, and knowledge on the domain from which the context is extracted by induction (e.g., generalizing "every bird flies" from $bird \sqsubseteq_K fly$).

Example 6 ($Triv$) *As the contextualized subsumption is isomorphic to the order on concepts (Theorem 2), it is possible to use the labelled concept lattice (see Figure 1.(b)) to study contextualized subsumption in context K_{Triv}. For instance, as concept 2 is smaller than concept 5 relation $c \sqsubseteq_{K_{Triv}} b \vee c$ stands, which is already true in $\mathcal{P}$. More generally, it can be seen that all valid subsumptions in $\mathcal{P}$ are retained in contextualized subsumption. Examination of the labelled concept lattice shows that the context adds new valid entailments between formulas: e.g., $c \sqsubseteq_{K_{Triv}} a \vee b$, because $2 \leq^c 4$.*

In the following sections, formal contexts will be used to formalize the content of an information system, and the concept lattice (or equivalently, the contextualized subsumption) will be used to organize things. It is the contextualized equivalence relation that gives so many names to things.

2.5 Feature Context

Logical languages contains usually infinitely many formulas, whose complexity is unbounded, which is a problem for algorithms that perform a search among formulas (e.g., learning, Ganter and Kuznetsov, 2001). For efficiency and readability of results, we restrict the search space of formulas to a finite subset $F \subseteq \mathcal{L}$ whose elements are called *features*. Features differ from attributes of standard formal contexts in three ways: (1) features belong to a fixed logical language and so, have a semantics, (2) features are automatically ordered according to the subsumption $\sqsubseteq$, and (3) a newly introduced feature can have a non-empty extent.

It is possible to extract a formal context, with F as the set of attributes, from the logical context: we call it the *feature context*. This context is not intended to be actually build from the logical context, but it is defined to allow reasoning about the logical context with a coarser grain than the full logic.

Definition 8 (feature context) *Let $K = (\mathcal{O}, \langle \mathcal{L}; \sqsubseteq \rangle, d)$ be a logical context, and $F_K \subseteq \mathcal{L}$ be a finite set of features, that may depend on K. The* feature context *of K is the formal context $K_F = (\mathcal{O}, F_K, I_{K_F})$, where $I_{K_F} = \{(o, x) \in \mathcal{O} \times F_K \mid d(o) \sqsubseteq x\}$. We also define* description features *for any object o by $D_{K_F}(o) = \uparrow_F d(o)$, where for any $f \in \mathcal{L}$, $\uparrow_F f = \{x \in F \mid f \sqsubseteq x\}$.*

The content of F_K is not strictly determined but depends on the context K. It should contain simple formulas subsuming logical descriptions (in K), frequently used formulas (in queries), and more generally, all formulas that users expect to see in answers. In fact, F_K acts as the vocabulary that a LIS uses in its answers. From the description given in Example 3, one can extract the following features (amongst others): `author contains "Ridoux"`, `year in 1950..2000`.

Feature concepts can be derived from a feature context as for ordinary logical contexts. Lemma 3 relates the Galois connections of logical and feature contexts.

Lemma 3 (logical vs. feature Galois connections) *Let $O \subseteq \mathcal{O}$, $X \subseteq F$,*

- $\sigma_{K_F}(O) = \uparrow_F \sigma_K(O)$
- $\tau_{K_F}(X) = \tau_K(\sqcap X)$.

Theorem 3 shows the existence of a mapping that approximates a logical concept in a feature concept and then define equivalence classes among logical concepts.

Theorem 3 (approximation of concept) *Let $(O, f) \in \mathcal{C}_K$ be a logical concept. The feature concept generated from O (intent: $\sigma_{K_F}(O)$) and the feature concept generated from f (extent: $\tau_{K_F}(\uparrow_F f)$) are in fact the same feature concept $(\tau_{K_F}(\uparrow_F f), \uparrow_F f)$, the smallest concept in $\mathcal{C}_{K_F}$ whose extent is larger than or equal to O.*

2.6 Sub-context

It is often useful to reason on a *sub-context* by restricting the set of objects and the set of features. For instance, the need for *views* has been recognized in databases (Ullman, 1988) (see also Definition 14).

Definition 9 (sub-context) *Given a* domain $D \subseteq \mathcal{O}$, *restricting the set of objects, and a* view $V \subseteq F$, *restricting the set of features, we define the* sub-context *of a feature context K_F by the formal context $K_F(D, V) = (D, V, I_{K_F} \cap (D \times V))$.*

Lemma 4 relates the Galois connections of feature contexts and sub-contexts.

Lemma 4 (feature vs. sub-context Galois connection) *Let $O \subseteq D$, and $X \subseteq V$,*

- $\sigma_{K_F(D,V)}(O) = \sigma_{K_F}(O) \cap V$,
- $\tau_{K_F(D,V)}(X) = \tau_{K_F}(X) \cap D$.

A domain can be specified by the extension $\tau_K(q)$ of a formula, i.e., the answers $\tau_K(q)$ to a query q. The specification of a view can be specified as the set of features subsumed by a query v, i.e., $\downarrow_F v = \{x \in F \mid x \sqsubseteq v\}$. E.g., in the logic presented in Example 3, the formula `(author contains "" ∨ title contains "concept" ∨ year in 1900..2000)` would select all features of attributes `author`, `title` and `year`, restricted to years in the last century, and titles that contain the word "concept".

3 Navigating and Querying in a Logical Context

3.1 Navigating vs. Querying

Information systems offer means for organizing data, and for navigating and querying. Though navigation and querying are not always distinguished because both involve queries and answers, we believe they correspond to very different paradigms of human-machine communication. In fact, the difference can be clarified using the intension/extension duality.

Navigation implies a notion of place, and of a relation between places (e.g., file system directories, and links or subdirectory relations). Through navigation, a user may ask for the content of a place, or ask for related places. The ability to ask for related places implies that answers in the navigation-based paradigm belong to the same language as queries. In terms of the intension/extension duality, a query is an *intension*, and answers are *extensions* for the content part, and *intentions* for the related places.

In very casual terms, we consider navigation with possibly "no road-map", i.e., no *a priori* overview of the country. Related places form simply the landscape from a given place as shown by a "viewpoint indicator". However, our proposal is compatible with any kind of *a priori* knowledge from the user.

With querying, answers are extensions only. A simulation of navigation is still possible, but forces the user to infer what could be a better query from the unsatisfactory answer to a previous query; i.e., infer an intension from an extension. This is difficult because there is no simple relation between a variation in the query, and the corresponding variation in the answer. The experience shows that facing a query whose extension is too vast, a user may try to refine it, but the resulting extension will often be either almost as vast as the former or much too small. In the first case, the query lacks of *precision* (i.e., number of relevant items in the answer divided by total number of items in the answer), whereas in the second case, the query *recall* (i.e., number of relevant items in the answer divided by number of relevant items in the system) is too low.

Godin et al. (1993) and Lindig (1995) have shown that Formal Concept Analysis is a good candidate for reconciliating navigation and querying. We follow this opinion, but we believe that care must be taken to make formal contexts as close to the description languages of the end-users, and we have proposed Logical Concept Analysis (LCA) where formal descriptions are logical formulas instead of being sets of attributes (Section 2.1).

Our goal in this section is to show how a form of navigation and querying can be defined, so that a user who knows neither the content of a Logical Information System, nor the logic of its descriptions, can navigate in it and discover the parts of the contents and the parts of the logic that are relevant to his quest. Note that a more expert user may know better and may navigate more directly to his goal, but since almost everybody has his shortcomings, the no-knowledge assumption is the safest one to do.

3.2 A Logical Information System

In this section, we will insist on navigation tools, and will delay the management of a logical context (e.g., creating, updating objects) until Section 4.

A LIS needs a user interface. We will formalize a shell-based interface, though this is not the most modern thing to do. This is because we believe that shell interfaces (like in UNIX or MS-DOS) are familiar to many of us, and because this abstraction level exposes properly the dialogue of queries and answers. A higher-level interface like a graphical one would hide it, whereas lower-level interfaces, like a file system, would expose irrelevant details. However, nothing prevents one to give a graphical interface to a LIS.

The shell commands are those of the UNIX shell, reinterpreted in the LIS framework. Main changes are the replacement of *paths* by formulas of $\mathcal{L}$ referring to concepts via mapping μ_K, and the use of contextualized subsumption $\sqsubseteq_K$. For the rest, commands have essentially the same effects. The correspondence between a UNIX file system and a logical information system is summarized in the following table.

UNIX shell	LIS shell
file	object
path	logical formula
absolute name of a file	object description
directory	formula/concept
root	formula $\top$/concept $\top^c$
working directory	working query/concept

Navigation commands `cd`, `ls`, and `pwd` are defined in Section 3.3; and the querying command `ls -R` is defined in Section 3.4. Creation and update commands `touch`, `mkdir`, `rm`, `mv`, and `cp` will be defined in Sections 4.1 and 4.2.

3.3 Navigating in a Logical Context

Once objects have been logically described and recorded in a logical context $K = (\mathcal{O}, \mathcal{L}, d)$, one wants to retrieve them. One way to do this is *navigating* in the context. As already said, this way of searching is particularly useful in a context where the logic or the content are unknown. The aim of navigation is thus to guide the user from a *current place* to a *target place*, which contains the object(s) of interest. For this, a LIS offers to the user 3 basic operations (the corresponding UNIX-like command names are placed between parenthesis): (1) to ask to LIS what is the current place (command `pwd`), (2) to go in a certain "place" (command `cd`), (3) to ask to LIS effective ways towards other "places" (command `ls`).

3.3.1 Places as formal concepts

In a hierarchical file system, a "place" is a directory. In our case, a "place" is a formal concept, which can be seen as a coherent set of objects (extent) and properties (intent) (see Definition 4). In large contexts, concepts cannot be referred to by enunciating either their extent or their intent, because both are generally too large. Formulas of the logic $\mathcal{L}$ can play this role because every formula refers to a concept through the labelling map μ (see Definition 6), and every concept is referred to by one or several formulas, which are often much concise than its intent. For instance in Figure 1, one can see that c is a concise name for the concept $(\{z\}, c \wedge (a \vee b))$.

We now describe the 3 navigation operations listed above. First of all, going from place to place implies to remember the current place, which corresponds to the working directory. In a LIS, we introduce the *working query*, wq, and the *working concept*, $wc := \mu_K(wq)$; we say that wq refers to wc. This working query is taken into account in the interpretation of most LIS commands, and it is initialized to the formula $\top$, which refers to the concept whose extent is the set of all objects. Command `pwd` displays the working query to the user.

The second navigation operation, command `cd`, takes as argument a query formula q saying in which place to go, and it changes the working query accordingly. We call l_{wq} (i.e., elaboration of wq) the mapping that associates to the query q a new working query according to the current working query wq. The query q can be seen as a *link* between the current and the new working query. Usually, `cd` is used to refine the working concept, i.e., to select a subset of its extent. In this case, the mapping l_{wq} is defined by $l_{wq}(q) := wq \sqcap q$, which is equivalently characterized by

$$\mu_K(l_{wq}(q)) =^c wc \wedge^c \mu_K(q) \text{ and } \tau_K(l_{wq}(q)) = \tau_K(wq) \cap \tau_K(q).$$

However, it is useful to allow for other interpretations of the query argument. For instance, we can allow for the distinction between *relative* and *absolute* queries, similarly to relative and absolute paths in file systems. The previous definition of the mapping l_{wq} concerns relative queries, but can be extended to handle absolute queries by $l_{wq}(/q) := q$, where '/' denotes the absolute interpretation of queries. This allows to forget the working query. We can also imagine less usual interpretations of queries like $l_{wq}(|q) := wq \sqcup q$. Finally, the special argument `..` for the command `cd` enables to go back in the history of visited queries/concepts. This works much like the "Back" button in *Web* browsers.

The last navigation operation, command `ls`, is intended to guide the user towards his goal. More precisely, it must suggest some relevant links that could act as queries for the command `cd` to refine the working query. These links are formulas of $\mathcal{L}$. A set of links given by `ls` should be finite, of course (whereas $\mathcal{L}$ is usually infinite), even small if possible, and complete for navigation (i.e., each object of the context must be accessible by navigating from $\top^c$).

3.3.2 Navigation Links

The following notion of refinement corresponds to the case where the elaboration mapping satisfies $l_{wq}(q) = wq \sqcap q$. To avoid to go in a concept whose extent is empty (a dead-end), we must impose the following condition on a link x: $\tau_K(wq \sqcap x) \neq \emptyset$. Furthermore, to avoid to go in a concept whose extent is equal to the extent of wq (a false start), we must impose this other condition: $\tau_K(wq \sqcap x) \neq \tau_K(wq)$. These conditions state that the extent of the new working query must be strictly between the empty set and the extent of the current working query. This characterizes relevant links (Lindig, 1995).

Now, as $\mathcal{L}$ is a too wide search space (it is often infinite), we will consider a finite set of features $F \subseteq \mathcal{L}$ in a context K in which links are selected.

Furthermore, we retain only greatest links (in the subsumption order) as they correspond to smallest refinement steps. Indeed, recall that each link is a suggestion to the user. Therefore, if the link `year in 1990..2010` is suggested, it is not worth suggesting the link `year = 2000`, because the former subsumes the latter. Following this principle, every conjunctive formula $x \sqcap y$ can be excluded from the search space for links, and so, of the set of features, because it is redundant with x and y. Thus, a way to build the set of features is to split object descriptions on outermost conjunctions (i.e., $a \wedge b$ is split into a and b, whereas $(c \wedge d) \vee e$ cannot be split this way).

However, to limit the number of links, it is useful to also extract abstracted form of these features, so as to allow a still more progressive navigation. For instance, the system would suggest the successive links `title`, `title contains "System"`, and then `title is "Logical Information Systems"`; instead of directly suggesting the latter. Only this latter feature appears explicitly in the description. The others are abstractions of it, and plays a role of "factorization" among links.

We now summarize this part by defining the set of links in a given context and working query.

Definition 10 (Navigation links) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. The set of* navigation links *for every working query $wq \in \mathcal{L}$ is defined by*

$$Link_K(wq) := Max_{\sqsubseteq}\{x \in F_K \mid \emptyset \neq \tau_K(wq \sqcap x) \neq \tau_K(wq)\}.$$

3.3.3 Local Objects and Navigation Completeness

As navigation aims at finding objects, command `ls` must not only suggest some links to other places, but also present the objects belonging to the current place, called the *objects of wq* or the *local objects*. We define a local object as an object that is in the current place, but in no place reachable through a link.

Definition 11 (local object) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. The set of* local objects *is defined for every working query $wq \in \mathcal{L}$ by*

$$Local_K(wq) := \tau_K(wq) \setminus \bigcup_{x \in Link_K(wq)} \tau_K(x).$$

There can be no local object, but there can also be several objects in some place. The less local objects there are, the better it is for navigation. More precisely, if two local objects have non-equivalent descriptions, it should be possible to make this difference appear in the links. Thus, the choice the user has to do is intentional (between one or several logical links) rather than extensional (between several objects). This idea is formalized as the *completeness* of navigation.

Definition 12 (navigation completeness) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. Navigation is* complete *in K if and only if for every working query $wq \in \mathcal{L}$, the following holds*

$$\forall o, o' \in Local_K(wq) : d(o) \equiv d(o').$$

This completeness can be guaranteed by ensuring that the set of features F_K is such that if two object have non-equivalent descriptions there exist a feature that is satisfied by one object and not by the other.

Theorem 4 (navigation completeness) *A necessary and sufficient condition for navigation completeness in a context K is that for every objects o, o'*

$$d(o) \not\equiv d(o') \Rightarrow \exists x \in F_K : d(o) \sqsubseteq x \Leftrightarrow d(o') \not\sqsubseteq x.$$

In the case where all objects have different descriptions, there is never more than one local object. This must be compared to *Web* querying where the number of objects returned in response to a query is generally large. This is because with navigation, non-local objects are hidden behind the intentional properties that enable to distinguish these objects. It is the end-user who selects an intentional property to reveal its content.

Another interesting thing to notice is that the working query can be, and is often, much shorter than the whole description of the local object (which is also the intent of the working concept), as in the following example where the first formula is contextually equivalent (in context Bib, see Example 3) to the second one for accessing the object.

```
cd author contains "Mineau" ∧ author contains "Missaoui" ≡
cd author is "Mineau, Missaoui" ∧ title is "The Representation of Semantic Constraints in Conceptual
Graph Systems" ∧ type is "InProceedings" ∧ year in 1997
```

3.3.4 Links and Views

Even if the set of links is restricted to relevant and greatest ones among features, it appears in practice that it is too large and heterogeneous. For instance, in the context *Bib* (see Example 3), the set of links is a mix of author names, title words, etc. Our idea is to abstract the set of all author names (features `author contains ...`) by a single feature `author` (meaning "the attribute `author` is defined"). This feature is not useful for selecting objects, but it is useful to select links about the attribute `author`. We call this kind of features *views* as they present the navigation from a "point of view". We introduce a working view wv, similar to the working query, under which links must be searched for. For instance, if the working view is (`author`), links will be author names. We take it into account in the definition of navigation links, where the relation $x \sqsubset wv$ denotes a strict subsumption (i.e., $x \sqsubseteq wv$ and $wv \not\sqsubseteq x$).

Definition 13 (Navigation links with views) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. The set of* navigation links *for every working query $wq \in \mathcal{L}$ and every working view $wv \in \mathcal{L}$ is defined by*

$$Link_K(wq, wv) := Max_{\sqsubseteq}\{x \in F_K \mid x \sqsubset wv, \emptyset \neq \tau_K(wq \sqcap x) \neq \tau_K(wq)\}.$$

As a link is a variation of the working query that restricts the current extent, one defines a *view* as a variation of the working view that restricts the set of links. Moreover, if a view is not also a link, it must subsume at least two links. Indeed, if some view hides only one link, it is worth presenting the link directly. Finally, we define a set of links and views where views can be understood as summaries of sets of links, whose selection by the user allows him to see these underlying links in a narrower view.

Definition 14 (Navigation links and views) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. The set of* navigation links and views *for every working query $wq \in \mathcal{L}$ and every working view $wv \in \mathcal{L}$ is defined by ($\|E\|$ denotes the cardinality of a set E)*

$$LV_K(wq, wv) := Max_{\sqsubseteq}\{x \in F_K \mid x \sqsubset wv, \quad \emptyset \neq \tau_K(wq \sqcap x) \neq \tau_K(wq) \text{ or } \|Link_K(wq, wv \sqcap x)\| \geq 2\}.$$

To summarize, the view-based variant of command `ls` takes as argument a view v, sets the working view to $l_{wv}(v)$ (where l_{wv} works similarly to l_{wq}), shows the local objects if they exist, displays each link or view x of the set $LV_K(wq, wv)$ along with the size of its selected extent $\tau_K(wq \sqcap x)$, and finally displays the size of the working extent $\tau_K(wq)$. Links and views are distinguished according to their cardinality compared to the size of the working query; views simply have the same size as the working concept, whereas links have strictly smaller sizes.

3.3.5 User/LIS Dialogue

We now show how commands `cd` and `ls` compose a rather natural dialogue between the user and LIS. The user can refine the working concept with command `cd`, and asks for suggested links and views with the command `ls`. LIS displays to the user relevant links for forthcoming `cd`'s, and relevant views for forthcoming `ls`'s. Commands `cd` (resp. links) are *assertions* from the user (resp. from LIS): "I want this kind of object!" (resp. "I have this kind of object!"). Commands `ls` (resp. views) are *questions* from the user (resp. from LIS): "What kind of object do you have?" (resp. "What kind of object do you want?"). It should also be noticed that both the user and LIS can answer to questions both by assertions and by questions.

Example 7 (*Bib*) *A complete example of a dialogue is given in Table 1. The left part of this table shows what is really displayed by our prototype, and the right part is an English translation of the dialogue. Notice that this translation is rather systematic and could be made automatic.* `(n)` *is the prompt for the n-th query from the user. On the 2nd query, the question of the user is so open, that LIS only answers by questions. On the 3rd query, the user replies to one of these questions (*`title`*) by an assertion; but on the 4th query, he sends back to LIS another of these questions (*`author`*) to get some relevant suggestions. On the 5th query, he just selects a suggested author,* `"Wille"`*, and then gets his co-authors on Concept Analysis with the 6th query. On the 7th query, he selects a co-author and finally finds an object at the 8th query.*

(1) pwd	(1) What is currently selected?
1	All objects.
(2) ls	(2) What do you have?
209 type	What kind of type do you want?
209 author	What kind of author do you want?
209 year	What kind of year do you want?
209 title	What kind of title do you want?
209 object(s)	209 objects are currently selected.
(3) cd title contains "Concept A"	(3) I want objects whose title contains "Concept A"!
(4) ls author	(4) What kind of author do you have (for this)?
1 author contains "Mineau"	I have 1 object with author "Mineau"!
1 author contains "Lehmann"	I have 1 object with author "Lehmann"!
1 author contains "Stumme"	I have 1 object with author "Stumme"!
1 author contains "Prediger"	I have 1 object with author "Prediger"!
3 author contains "Wille"	I have 3 objects with author "Wille"!
4 object(s)	4 objects are currently selected.
(5) cd author contains "Wille"	(5) I want objects with author "Wille"!
(6) ls	(6) What kind of author do you have (now)?
1 author contains "Mineau"	I have 1 object with author "Mineau"!
1 author contains "Lehmann"	I have 1 object with author "Lehmann"!
1 author contains "Stumme"	I have 1 object with author "Stumme"!
3 author contains "Wille"	What kind of author "Wille" do you want?
3 object(s)	3 objects are currently selected.
(7) cd author contains "Mineau"	(7) I want objects with author "Mineau"!
(8) ls	(8) What do you have?
#200 Mineau, Stumme, Wille. Conceptual Structures Represented by Conceptual Graphs and Formal Concept Analysis. INPROC, 1999.	
1 object(s)	1 object is currently selected.
(9) pwd	(9) What is currently selected?
author contains "Wille" $\wedge$	Objects with authors "Wille"
author contains "Mineau" $\wedge$	and "Mineau",
title contains "Concept A"	and whose title contains "Concept A".

Table 1: Example of User/LIS Dialogue in context *Bib*.

3.4 Querying a Logical Context

Extensional queries, as in data-bases or some *Web* browsers like Google, can be submitted to a logical information system using the `-R` option with command `ls`. The answer to query `ls -R q` is simply $\tau_K(l_{wq}(q))$, i.e., the extent of the concept referred to by $l_{wq}(q)$ (see Section 3.3).

```
(1) ls -R /title contains "Logic" ∧ ¬ title contains "Concept" ∧ year in 1990..1995
#3 Gaines. Representation, discourse, logic and truth: situating knowledge technology. INPROC,
1993.
#2 Sowa. Relating diagrams to logic. INPROC, 1993.
#72 Van den Berg. Existential Graphs and Dynamic Predicate Logic. INPROC, 1995.
3 object(s)
```

4 Creating and Updating a Logical Context

Whereas much has been said on the construction of concept lattices (Kuznetsov and Objedkov, 2001), the construction of contexts is often left in the background. The construction process can fall into two categories: off-line and on-line. In the off-line case, the context is built once for all after the data have been gathered and the problem is to find an object description language appropriate to the intended analysis. The typical application of this category is the analysis of surveys. In the on-line case, the context is built progressively along the arrival of data and a problem is to properly describe new objects at the time they arrive. Information systems are the typical application of this category, but this is not the mainstream approach to using Concept Analysis.

Hypothesis 1 (on-line construction) *We consider here only the on-line case, as we focus on information systems. For each new piece of data that arrives, an object is created, and added to the context, with this piece of data as content.*

We propose that the description given to an object is two-parts. The first part, the *intrinsic description*, is automatically extracted from the object content, and depends on the kind of content and on the logic of the application. For instance, let us consider that objects are incoming e-mail messages. In this application, the

building of the context is clearly on-line; and possible components of the intrinsic description are the `from`, `to`, and `subject` fields.

The second part, the *extrinsic description*, is manually assigned by users according to personal intentions and preferences. We must consider there are no known rules to infer extrinsic properties, as if the contrary holds they could be integrated in the intrinsic description. In a usual e-mail application, extrinsic properties are managed by storing e-mail messages in different folders according to personal needs. However, extrinsic properties need not be organized in a hierarchical relation as folders often are.

4.1 Creating a Logical Context

The objects we want to represent in a context are often defined in the "world" by a *content*. We want to describe these objects both by an *intrinsic* properties (automatically extracted from their contents), and by *extrinsic* properties (manually assigned by users). All these elements are the *context data*, from which a logical context can be built.

Definition 15 (context data) Context data *are defined as a 6-tuple* $D = (\mathcal{O}, \mathcal{C}, \mathcal{L}, c, d_i, d_e)$, *where*

- $\mathcal{O}$ *is a set of* objects,
- $\mathcal{C}$ *is the domain of* contents,
- $\mathcal{L}$ *is a* logic,
- $c \in \mathcal{O} \to \mathcal{C}$ *maps every object to its content,*
- $d_i \in \mathcal{C} \to \mathcal{L}$ *extracts an* intrinsic description *from every content,*
- $d_e \in \mathcal{O} \to \mathcal{L}$ *maps every object to its* extrinsic description.

The building of a context from its data consists in keeping the set of objects and the logic, and composing a description that maps every object to its intrinsic description "plus" its extrinsic description. This "plus" denotes an *update* operation $\diamond$ that we present in more details in Section 4.2.

Definition 16 (context building) *Let* $D = (\mathcal{O}, \mathcal{C}, \mathcal{L}, c, d_i, d_e)$ *be context data. The* context built from data D *is defined as*

$$K(D) = (\mathcal{O}, \mathcal{L}, d), \text{ where for every } o \in \mathcal{O}, d(o) = d_i(c(o)) \diamond d_e(o).$$

Example 8 ($E-Mail$) *In this section, we consider that objects are e-mail messages. An example of a simplified e-mail message content is:*

```
From: Alice@paris.fr
To: Bob@berlin.de, Chloe@madrid.es
Date: 2 May 2002, 14:52
Subject: Hello world!

When do you come in Paris?

See you,
Alice
```

From such a content, an intrinsic description can be extracted and formulated in logic $\mathcal{P}_v$ *(see Example 3), while retaining only fields* `from`, `to`, *and* `subject`. *This is only a matter of choice, as other fields could be taken into account. In particular, the message body would certainly be useful in a real application, but it is not necessary to our explanations. The intrinsic description we obtain from the above content is the following formula.*

```
from is "Alice@paris.fr"
∧ to is "Bob@berlin.de, Chloe@madrid.es"
∧ subject is "Hello world!"
```

Then, the user (e.g., Bob) can add his personal comments by formulating an extrinsic description. For instance, the extrinsic description

`personal` $\wedge$ $\neg$`spam`

means the message is "personal" (opposite of "professional"), and is not a "spam". The expected result of updating the intrinsic description with the extrinsic description is

```
from is "Alice@paris.fr"
∧ to is "Bob@berlin.de, Chloe@madrid.es"
∧ subject is "Hello world!"
∧ personal ∧ ¬spam
```

The LIS shell commands for creating new entries in a context are `mkdir` and `touch`. Command `mkdir` creates a new feature. For instance, `mkdir subject begins with "H"` introduces a finer feature than those that are obtained by simply splitting conjunctions. Command `touch` simply creates an object with empty content at some place designated by a formula. However, objects are normally created by applications, which give them a content (and so, an intrinsic description) and asks users for the extrinsic description.

4.2 Updating a Logical Context

From the Definition 15 of context data, a logical context can be updated in 4 ways: (1) addition of an object with its initial content and extrinsic description (commands `touch`, `cp`, and applications), (2) update of the content of an object (applications), (3) update of the extrinsic description of an object (command `mv`), and (4) deletion of an object (command `rm`).

The only difficulty that arises from these operations is the *update* operation, which is used to change an extrinsic description in an incremental way (i.e., without completely redefining it), and to compose intrinsic and extrinsic descriptions. A naive solution would be to manually change extrinsic descriptions, and to directly integrate them in intrinsic descriptions. However, this has two important drawbacks. First, the intrinsic description changes with the contents and so, it is not a persistent support for extrinsic properties. Second, practice shows that one often wishes to express an update operation for a set of objects in a single command: for instance, add property "personal" (`personal`) to every e-mail message sent from "Alice" (`from contains "Alice"`). We give now a specification of such an operation, as it is given by Herzig and Rifi (Herzig and Rifi, 1999). This puts constraints on what a logical update operation should be.

Definition 17 (update) *Let $\mathcal{L} = (L, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$ be a logic. The result of* updating *a description $d \in L$ by an entry $e \in L$ is the result of an operation $d \diamond e$, which must satisfy the following postulates:*

1. *(HR) $d \diamond e \sqsubseteq e$;*
2. *(HR) $d \sqcap e \sqsubseteq d \diamond e$;*
3. *(HR) $d \diamond \top \equiv d$;*
4. *(HR) $d \not\sqsubseteq \bot$ and $e \not\sqsubseteq \bot$ implies $d \diamond e \not\sqsubseteq \bot$;*
5. *(HR) for every $d' \equiv d$, $d' \diamond e \equiv d \diamond e$;*
6. *(HR) for every $e' \equiv e$, $d \diamond e' \equiv d \diamond e$;*
7. *(HR) for every $d_1, d_2 \in L$ such that $d \equiv d_1 \sqcup d_2$, $d \diamond e \equiv (d_1 \diamond e) \sqcup (d_2 \diamond e)$;*
8. *(involution) $(d \diamond e) \diamond e \equiv d \diamond e$.*

Postulate 1 means that the entry must always be taken into account, and so, satisfied in the result. Postulate 2 adds that everything satisfied in the result comes from either the description d or the entry e. In the case where the entry is the tautology, i.e., brings no information, postulate 3 states that the description must be kept unchanged. Postulate 4 forces the result to be consistent, unless the description or the entry is already inconsistent. Postulates 5 and 6 mean that results must not depend on the syntax of formulas. Finally, postulate 7 says that operation $\diamond$ must be distributive for disjunction, and postulate 8 says that it must be an involution.

Herzig and Rifi (1999) present an update operation for propositional logic with dependencies between atoms, denoted by $WSS \downarrow^{dep}$, that satisfies all postulates in Definition 17. In logic, dependency between atoms is subsumption.

Example 9 (*Bib*) *Logic $\mathcal{P}_v$ is a propositional logic, whose usual atoms are replaced by valued attributes. This means that we must consider dependencies between such atoms. For instance, the atom* `from is "Alice@paris.fr"` *implies the atom* `from contains "Alice"`*, contradicts the atom* `from is "Bob@berlin.de"`*, but is independent from the atom* `subject contains "Hello"`*.*

The LIS shell commands `rm`, `mv` and `cp` perform LIS updates. Command `rm q` suppresses the local object of $l_{wq}(q)$ (if it exists, see Section 3.3.3). Command `mv` moves a file from a place to another: `mv q e` moves the local object o of $l_{wq}(q)$ in concept $\mu(d(o) \diamond e)$. Similarly, `cp q e` creates a new copy of the local object in the same concept. With option `-r`, every object of the extent of $l_{wq}(q)$ is concerned, instead of only the local object.

```
(1) cd /from contains "Alice"
(2) mv .  personal
(3) rm -r /spam
```

The move command (line 2) adds the feature `personal` to the local object of the working query `from contains "Alice"`. Note that feature `personal` is clearly extrinsic. The remove command (line 3) deletes all objects described as a spam.

Contents can also be changed by applications. This changes indirectly descriptions (their intrinsic parts), but the ensuing reorganization of the formal concept lattice is automatic and transparent. In fact, it costs not so much since the concept lattice is not actually represented (see Section 6).

5 Data-mining, automated updating and learning

Previous sections 3 and 4 present the core operations of a LIS. More operations can be defined. Some are derived from core operations, like a form of data-mining, and others are disjoint from the core but can be added to it.

5.1 Data-mining

The definition of links (see Definitions 3.3.2 and following) is a specific case of Knowledge Discovery in a formal context. It can be generalized to recover more classical KD operations like machine-learning through the computation of necessary or sufficient properties (modulo some confidence), or data-mining through association rules. Indeed, CA has been often applied in domains such as *data-analysis*, *data mining*, and *learning*.

Data-analysis consists in structuring data in order to help their understanding. These data are often received as tables or relations and structured by partitions, hierarchies, or lattices. With CA, formal contexts (binary relations between objects and attributes) are structured in concept lattices (Ganter and Wille, 1999). This is applied for instance in software engineering for configuration analysis (Krone and Snelting, 1994). *Data-mining* is used to extract properties from large amounts of data. These properties are association rules satisfied (exactly or approximately) by the data. This is analogous to implications between attributes in FCA (see Ganter and Wille, 1999, p. 79), and to contextualized subsumption in LCA (see Section 2.4). *Unsupervised learning* is similar to data-analysis in the sense that one tries to discover some properties, and to understand some data, whereas *supervised learning* is similar to data-mining as some rules are searched to explain a target property using known properties. For instance, Kuznetsov applied CA to the learning of a positive/negative property from positive and negative instances (Kuznetsov, 1999).

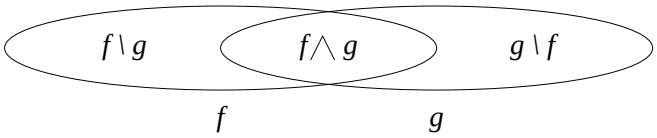
The aim of this section is to show that these features of Knowledge Discovery (KD) can be incorporated in LIS, and how.

A context K plays the role of a theory by extending the subsumption relation and enabling new entailments (e.g., $bird \sqsubseteq_K fly$ when every bird flies in the context). All these contextual entailments are gathered with logical entailments to form the contextualized logic, which is thus a means for extracting some knowledge from the context. Two kinds of knowledge can be extracted: knowledge about the context by deduction ("Every bird of this context *do* fly"), and knowledge about the domain (which the context belongs to) by induction ("Every bird of the domain *may* fly").

Concept lattices produced by data-analysis are isomorphic to contextualized logics (see Theorem 2). Associations rules produced by data-mining or supervised learning match the contextualized subsumption relation, possibly qualified by a confidence defined by $conf(f \sqsubseteq_K g) = \frac{|\tau_K(f) \cap \tau_K(g)|}{|\tau_K(f)|}$.

Considering two properties $f, g \in \mathcal{L}$, their contextual relation is determined by the sizes of 3 sets of objects: $\pi^l_K(f,g) := |\tau_K(f) \setminus \tau_K(g)|$, $\pi^c_K(f,g) := |\tau_K(f) \cap \tau_K(g)|$ and $\pi^r_K(f,g) := |\tau_K(g) \setminus \tau_K(f)|$. For instance,

f contextually entails g if and only if $\pi^l_K(f,g) = 0$, f and g are contextually separated if and only if $\pi^c_K(f,g) = 0$, or x is a link of wq (see Section 3.3.2) if and only if $\pi^c_K(x, wq) \neq 0$ and $\pi^r_K(x, wq) \neq 0$. Note that the superscripts, l, c, and r refer to the left, center, and right part of the following Venn diagram:



So, the procedure that computes links for navigating in a LIS can be generalized to compute necessary or sufficient conditions, association rules, and links, only by specifying constraints on π^c_K, π^r_K, and π^l_K.

5.2 Context Maintenance based on Learning

A context associates to objects a description that combines automatically extracted properties (*intrinsic*) and manually assigned ones (*extrinsic*) (cf. Section 4). The extrinsic properties are expressed by users according to intentions that are often subjective and changing, and determine the classification and retrieval of objects. So, we believe it is important to assist users in this task through the automatic suggestion of extrinsic properties to be assigned and even the discovery of rules to automate these assignments. The principle is to learn from the relationship between extrinsic and intrinsic descriptions of existing objects the extrinsic description of a new object whose intrinsic description is computed from its content. Because of the changing nature of users' intentions, the assistance given in the incremental building of a logical context must be interactive. We present formal principles, and an application to the classification of e-mail messages. Proofs can be found in (Ferré and Ridoux, 2002).

5.2.1 Induction through Associative Concepts

Let us consider the situation where a new object o^* is added to a logical context $K = (\mathcal{O}, \mathcal{L}, d)$ along with an intrinsic description $d^*(o^*)$ to form a new context $K^* = (\mathcal{O} \uplus \{o^*\}, \mathcal{L}, d^*)$ with $d^*(o) = d(o)$ for all $o \in \mathcal{O}$. Our aim is to induce from the old context K a set of extrinsic properties $Ind_{K_F}(o^*) \subseteq F$ for the new object.

We first define *associative concepts* as the concepts of K when one considers only description features $D_{K_F}(o^*)$.

Definition 18 (associative concept) *A non-empty concept of the sub-context $K_F(\mathcal{O}, D_{K_F}(o^*))$ is called an* associative concept *of o^* in K_F. The set of all such associative concepts is denoted by $AC_{K_F}(o^*)$.*

$AC_{K_F}(o^*)$ organizes the feature context K_F in a concept lattice (where the empty concept is missing) that is less finely detailed than $\mathcal{C}_{K_F}$. However, this coarser concept lattice is relevant to the features of o^*. Conversely, the finer details in $\mathcal{C}_{K_F}$ cannot be expressed with the features of o^*.

Then, an *induced feature* can be defined as a feature that contextually subsumes the intent of some associative concepts.

Definition 19 (induced property) *We say a feature x is an induced property if and only if there exists an associative concept $c \in AC_{K_F}(o^*)$, such that $\sqcap int(c) \sqsubseteq_K g$ or, equivalently, $ext(c) \subseteq \tau_K(x)$. $Ind_{K_F}(o^*)$ denotes the set of all induced properties of o^* in K_F.*

Intuitively, an associative concept c of a new object o^* is an already existing concept (for previous objects in K) that has some similarity with the description of o^*. When $x \in Ind_{K_F}(o^*)$ is induced from an associative concept c, $ext(c)$ is the *support* of the induction, and $int(c)$ is the *explanation*. A given associative concept can induce several features; and a given feature can be induced by several associative concepts, and so, have several explanations. Induced features that do not belong to description features are called *expected features*, which are the suggested features to users as extrinsic properties.

5.2.2 Experimentation

The aim of this section is to present through experimentations the kind of interactions that help a user to assign extrinsic properties to incoming objects, and to gradually automate these assignments (e.g., for filtering spams).

Filtering spams We consider here the assisted filtering of spams. The following display shows the initial description of a new (non-solicited) e-mail message, with its expected features. The context on which the induction of expected features is based is made of 200 e-mail messages.

```
Current description:
   from is "hh2732774@dtcom.net" ∧
   to is "undisclosed-recipients" ∧
   subject is "earn money without a job!"

Expected features:
28 spam
     <- from contains "net" ∧ to is "undisclosed-recipients"
     <- to is "undisclosed-recipients"
     <- from contains "net"
     <- subject contains "earn" ∧ subject contains "money"
2 to contains "irisa"/ to contains "fr"/ from contains "com"
     <- subject contains "earn" ∧ subject contains "money"
...
```

We can see that, whereas the `from` and `subject` field are new, several features are induced. This is possible because message fields are split in more or less general words, which enables to find common features between the new object and existing ones: e.g., `from contains "net"`, `subject contains "money"`, `to is "undisclosed-recipients"`.

We also see that the above message is well-recognized as a spam, and this feature is even the most strongly induced one with several explanations and a weight of 28 supporting objects. These explanations suggest some rules such as "every e-mail message sent to `undisclosed-recipients` is a spam", or "every e-mail message whose subject contains words `earn` and `money` is a spam".

It is up to the user to validate them, and make them automatic, or to be cautious, and only consider them as hints.

Figure 2 shows the filtering rate of spams during the incremental building of an e-mail context. The dashed line represents the rate of well-classified messages (as spam or non-spam) at the n-th insertion. The solid line represents the rate of classified messages (well or not). So, the part between the two lines represents badly classified messages (e.g., spam classified as non-spam), and the part above the solid line represents non-classified messages. This plot shows that after a transient phase of about 50 messages, the rate of well classified messages steadily reaches 85%, and there are nearly no bad classification. only one).

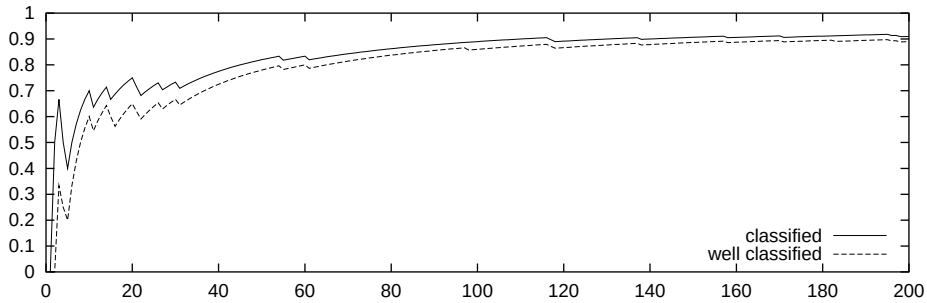


Figure 2: Filtering spams

The rates in Figure 2 are accumulated from the start of the experiment: $classified(n) = \frac{total\ classified(n)}{n}$. So they take into account the bad rates of the transient phase. The pseudo-instantaneous rates, $classified(n) = \frac{total\ classified(n) - total\ classified(n-\delta t)}{\delta t}$ are over 90% after the 80^{th} message for a time window (δt) of 50 messages (average number of messages per week).

Classifying e-mail messages The second application is a variant of the first one in which keywords are not limited to two values. We classify e-mail messages in about 20 non-exclusive categories such as teaching,

research, spam, call-for-paper, and so on. Note that these categories were not fixed a priori, but appeared only when required by the meaning of incoming messages. Thus, the vocabulary of categories remains open for ever.

Figure 3 shows the results of this experiment. The "automatic" line shows the rate of automatic classification using rules that are suggested by the system and accepted by the user (40 rules, including 15 for spams). The "suggested" line shows the rate of correct suggested classification. It tends to decrease simply because the sum of the two rates must be less than 1. Both rates are measured in number of features. The solid line shows the sum of the two rates.

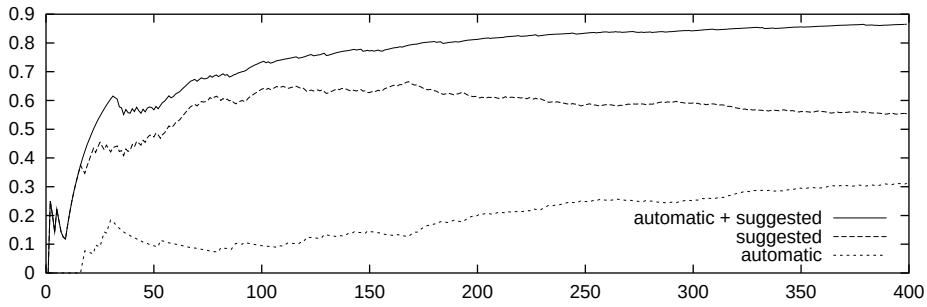


Figure 3: Classifying e-mail messages

Note again that these rates are cumulative; the instantaneous rates of "automatic+suggested" is constantly over 85% after the 100^{th} message for a time window of 50 messages.

6 Implementation of a LIS

Logic information systems are based on two choices: Concept Analysis and the use of an arbitrary logic. Both choices ask for performance questions as the complexity of concept lattices is exponential, and using logical descriptions implies the introduction of a theorem prover. However, theorem proving may be costly, e.g., exponential time for propositional logic.

6.1 Concept analysis without concept lattice

We first present the internal representation of a logical context we have chosen for our prototype; we also describe operations on this representation. Then, we give space complexity for the internal representation, and time complexity for the operations. These complexities are based on hypotheses, justified by experiments.

6.1.1 Internal Representation of a Logical Context

In a logical context $K = (\mathcal{O}, \mathcal{L}, d)$, we are mainly interested in navigation and querying. From their definition in Sections 3.3 and 3.4, it appears that all needed elementary operations are:

- $\sqsubseteq$: *subsumption* test between two formulas,
- τ_K: *extent* of a formula, which is based on subsumption $\sqsubseteq$ and on the set of objects $\mathcal{O}$,
- $\cap$: *set intersection* (between two extents),
- $Max_{\sqsubseteq}$: *selection* of maximal formulas w.r.t. $\sqsubseteq$, from a set of formulas.

Three of these operations use the subsumption, i.e., call the theorem prover of the logic. As its complexity can be high in some logics, it is preferable to limit its use as much as possible. In particular, the computation of extents is intensively used in the search for links, whereas each computation of an extent means as many subsumption tests as the number of objects.

We propose to store subsumption relations between features, and between features and object descriptions, in what we call a *logical cache*. We also propose to cache extents of features, which are intensively used for navigation. Thus, a logical context K is stored as a directed acyclic graph whose nodes are logical formulas

(features F_K and object descriptions $d(\mathcal{O})$), equipped with their extent, and whose arcs are subsumption relations between these formulas (arcs deducible by reflexivity and transitivity are not represented). This representation is the *Hasse diagram* of the partially ordered set $\langle F_K \cup d(\mathcal{O}); \sqsubseteq \rangle$.

Note that this representation is different from the concept lattice, which is usually used in systems based on concept analysis (e.g., Godin, Missaoui, and April, 1993; Cole and Stumme, 2000). We think the logical Hasse diagram is more appropriate to logical navigation for several reasons:

- the order used to compare a potential link to the working view and to select maximal links is the subsumption $\sqsubseteq$, and not the order on concepts $\leq^c$,
- the search space for navigation links is restricted to the features subsumed by the working view without testing $x \sqsubset wv$; and similarly to select maximal features without actually checking subsumption,
- the number of concepts can be exponential with the number of objects, whereas the set of features is sufficient for navigation, and its size is linear with the number of objects (see Section 6.1.3),
- the order $\leq^c$ on concepts changes when objects are added, contrary to the order $\sqsubseteq$ on formulas.

The principal operations we consider on this logical cache are:

insertion of a formula: a new feature or object description is inserted in the graph and connected to other formulas according to the subsumption relation;

addition of an object: a new object is placed on the node representing its description, and extents of existing nodes are updated;

search for the maximal features satisfying some property: this is used to search for navigation links and views.

These operations are sufficient to implement the shell commands we presented for navigation and querying.

6.1.2 Experiments

A prototype of a Logical Information System has been built for experimentation purpose. It has been implemented in Prolog as a generic system in which a theorem-prover and a syntax analyzer can be plugged-in for every logic used in descriptions. It is not meant to be efficient, though it can handle several thousand entries. Contrary to other tools based on concept analysis, it does not create the concept lattice. It only manages a Hasse diagram of the features used so far.

For the ICCS *Bib* context (see Example 3), the Hasse diagram has 954 nodes and 2150 arcs. In the experiments reported in this article, all response times are shorter than 1 second. In other experiments with a full-sized *Bib* context, i.e., all fields are represented and there are several thousand bibliographical references, the Hasse diagram has an average of 15 nodes per object, 3 arcs per node, and a height of about 5. This experiment and others support the idea that the number of features per object, f, is nearly constant for a given application; e.g., f is about 60 in *Bib* contexts. This has a positive implication on the complexity of LIS operations, since under this hypothesis their time complexity is either constant, or linear with the number of objects (see below).

6.1.3 Theoretical and Practical Complexity

The two important parameters for determining complexities are (1) the number of objects, n, and (2) the number of features per object, f (there are other parameters, but they are bounded by f). The space complexity of the Hasse diagram is in $\mathcal{O}(f^2 n)$. Among the three above operation, only the insertion of a formula uses the subsumption: the number of call to $\sqsubseteq$ is in $\mathcal{O}(fn)$. About set operations, the time complexity is either in $\mathcal{O}(f^2)$ (insertion of a formula, and addition of an object), or in $\mathcal{O}(fn)$ (search for links and views).

Our experiments have shown that the number of features per object f is nearly constant for a given application (see Section 6.1.2). This is explained by the fact that features are extracted from object descriptions automatically and without considering other objects. The consequence for complexities is that every operation is either constant, or linear with the number of objects. This means that navigation and querying in a LIS are tractable.

6.2 Logics for LCA and LIS

We present in this section how the generic scheme that takes a logic as a parameter can be instantiated.

6.2.1 Principles

Using logics as schemas in data-bases implies that end-users or system administrators will have to define and implement logics. Clearly, this is out of reach for most of them. In order to make our principles practicable anyway, we have designed a framework for specifying and implementing logics. This framework is based on what we called *logic functors*. Using them, defining and implementing a logic consists merely in combining parameterized logics. Each logic functor consists in a logic component, e.g., propositional logic or interval comparison. Functors can be composed to form new logics, e.g., propositional logic on intervals.

Each functor is implemented as a parameterized theorem prover. Our goal is that the theory and theorem prover of a combination of functors result from a systematic combination of the theory and theorem prover of each functor.

All functors and their compositions implement a common interface which corresponds to the 6-tuple of Definition 1. This makes it possible to program generic applications that can be instantiated with a logic component. Conversely, customized logics built using the logic functors can be *embedded* in an application that respects this interface.

The whole framework development is geared towards manipulating logics as lattices. So, subsumption is considered as a relation between formulas, and we study the conditions under which this relation is a partial order.

Our idea is to consider that a logic interprets its formulas as functions of their atoms. By abstracting atomic formulas from the language of a logic we obtain what we call a *logic functor*. A logic functor can be applied to a logic to form a new logic. For instance, if propositional logic is abstracted over its atomic formulas, we obtain a logic functor called *prop*, which we can apply to, say, a logic on intervals *interv*, to form propositional logic on intervals, *prop*(*interv*).

6.2.2 Logics and logic functors

We present the logic functor structure. More details can be found in (Ferré and Ridoux, 2001), particularly on the conditions for composability.

Definition 20 (logic) *A logic L is a triple (AS_L, S_L, P_L) where AS_L defines the abstract syntax of formulas of L, S_L defines their semantics, and P_L defines their interface.*

S_L is a pair $(I_L, \models_L)$ where

- *I_L is the interpretation domain of the formulas of L, and*
- *$\models_L \in \mathcal{P}(I_L \times AS_L)$ is the satisfaction relation between interpretations and formulas; $i \models_L f$ means that i is a model of f. We write $M_L(f) = \{i \in I_L \mid i \models_L f\}$ the set of all models of a formula f.*

P_L is a 5-uple $(\sqsubseteq_L, \sqcup_L, \sqcap_L, \top_L, \bot_L)$ where

- *$\sqsubseteq_L \in \mathcal{P}(AS_L \times AS_L)$ is the entailment relation,*
- *$\sqcup_L, \sqcap_L \in AS_L \times AS_L \to (AS_L \cup \{undef\})$ are conjunction and disjunction, and*
- *$\top_L, \bot_L \in AS_L \cup \{undef\}$ are the tautology and contradiction for L.*

All logics built with logic functors present the same interface ($\sqsubseteq_L$,$\sqcup_L$,$\sqcap_L$,$\top_L$,$\bot_L$), plus other operations like updating (operation $\diamond$ in Section 4.1), parsing and printing. So, they implement the same abstract data-type. The 5 logic operations considered here correspond to a minimal requirement about logics; the ability to test entailment, to build new formulas using conjunction and disjunction, and to determine if a formula is a tautology or a contradiction. A logic may have more connectives, but they will appear in its abstract syntax, not in its interface.

In order to simplify the presentation, we will only consider operations $\sqsubseteq_L$, $\sqcap_L$, and $\top_L$ in the sequel.

The definition of $\sqcap_L$ needs not be total. Similarly, $\top_L$ needs not be defined. Moreover, there is no *a priori* relation between S_L and P_L. So, we need a notion of completeness and consistency that takes into account partial definitions.

Definition 21 (completeness and consistency) *Let L be a logic, the operations of its interface P_L, i.e., $\sqsubseteq_L$, $\sqcap_L$, and $\top_L$, are consistent (resp. complete) w.r.t. a semantics S_L, if and only if for all formulas $f, g \in AS_L$ we have*

- $f \sqsubseteq_L g \Longrightarrow M_L(f) \subseteq M_L(g)$ *(resp.* $M_L(f) \subseteq M_L(g) \Longrightarrow f \sqsubseteq_L g$*),*
- $\top_L$ *is always consistent (resp.* $\top_L \neq undef \Longrightarrow M_L(\top_L) = I_L$*),*
- $f \sqcap_L g \neq undef \Longrightarrow M_L(f \sqcap_L g) \subseteq M_L(f) \cap M_L(g)$
  (resp. $f \sqcap_L g \neq undef \Longrightarrow M_L(f \sqcap_L g) \supseteq M_L(f) \cap M_L(g)$*),*

An interface P_L is consistent (resp. complete) w.r.t. a semantics S_L if and only if each of its operation is consistent (resp. complete).

By this definition, a completely *undef*ined interface is trivially complete and consistent, but it is useless too. So, the game of designing a new logic is to make it defined enough to be useful, but still complete and consistent.

Definition 22 (logic functor) *Assuming $\mathbb{L}$, $\mathbb{AS}$, $\mathbb{S}$, and $\mathbb{P}$ the collections of all logics, abstract syntax, semantics, and interface, a logic functor $F : \mathbb{L}^n \to \mathbb{L}$ is a triple (AS_F, S_F, P_F) defined as follows:*

- $AS_F : \mathbb{AS}^n \to \mathbb{AS}$ *such that* $AS_{F(L_1,\ldots,L_n)} = AS_F(AS_{L_1}, \ldots, AS_{L_n})$,
- $S_F : \mathbb{S}^n \to \mathbb{S}$ *such that* $S_{F(L_1,\ldots,L_n)} = S_F(S_{L_1}, \ldots, S_{L_n})$,
- $P_F : \mathbb{P}^n \to \mathbb{P}$ *such that* $P_{F(L_1,\ldots,L_n)} = P_F(P_{L_1}, \ldots, P_{L_n})$,

By convention, a logic will be considered as a logic functor of arity 0.

Logic functors are used as follows. S_L describes the semantics; it acts as a specification. P_L implements an interface; its description must be constructive enough, so that it leads directly to a program. A part of the interface describes how the concrete syntax is parsed and printed (remember that AS_L is only the abstract syntax). The other part offers logic operations. The user composes a logic by applying logic functors to logics, say $F(L_1, \ldots, L_n)$, and a *logic composer* takes such an expression and produces automatically the concrete implementation of the logic by gluing together the concrete implementations of F, L_1, ..., and L_n. This results in a software component, with a formally specified interface, that can be plugged in any software system that assumes the same interface.

This methodology leads to designing a library of logic functors for describing objects of LIS: a unary propositional functor, whose main quality is to make a total logic out of a partial one ($prop(partial) = complete$), several nullary logic functors for concrete domains like strings and intervals, several n-ary logic functors for combining descriptions, and an auto-epistemic logic functor for representing (un)completeness of knowledge.

It is worth insisting on the auto-epistemic logic functor. Very often, users expect that a description `author is "Smith"` implies ¬ `author is "Jones"`. However, this is false in standard propositional logic. One needs a form of Closed World Assumption or a kind of Negation as Failure to prove that. However, to be used in a LIS a logic must be monotonic (because its subsumption relation must form a lattice). $\mathcal{ONL}$ (a.k.a. "All I Know" Levesque, 1990) is an epistemic modal logic that permits to express that a formula tells the whole truth. It is a variant of $\mathcal{ONL}$ that we are using in LIS (Ferré, 2001); and its functor is called *aik*. Since it is very frequent to have to express absolute knowledge, the idiom $prop(aik(prop(\ldots)))$ has become a mandatory prefix of the logics used in LIS.

The principle of composing logic functors has been implemented in a prototype. It reads logic specifications such as $prod(prop(atom), prop(interv))$ and produces automatically a printer, a parser, and a theorem-prover. This example means that logic formulas are products of a proposition on atoms and a proposition on intervals. The prototype reads such a constructed logic and builds a theorem-prover for it by instantiating the theorem-prover associated to each logic functor at every occurrence where it is used. The prototype, each functor implementation, and the resulting implementations are written in λProlog.

Coming back to the bibliography example of the introduction, we construct a dedicated logic with logic functors as follows:

$$prop(aik(prop(sum(atom, valattr(sum_n(interv, string, \ldots)))))).$$

which means propositional logic on epistemic modal formulas on propositional logic on a sum (i.e., a mix) of atomic formulas and valued attributes whose values are themselves made of interval formulas, string formulas, etc. According to the theory of logic functors, the following theorem holds.

Theorem 5 (logic is ok) *The logic*
$L = prop(aik(prop(sum(atom, valattr(sum_2(interv, string))))))$
is such that P_L is total, bounded, and consistent and complete in all five operations w.r.t. S_L.

7 Conclusion

We have presented the specifications of a Logical Information System based on (Logical) Concept Analysis. It is based on Concept Analysis and is generic w.r.t. a logic for describing objects. In this framework, navigation/querying and creation/updating can be seamlessly integrated.

In this way, standard commands of a file system shell can be mimicked in a logical context. However, a simple generalization of the definition of links forms a framework in which operations of data-analysis or data-mining can be expressed. Using this framework, purely symbolic navigation as well as statistical exploration can be integrated smoothly as variants of the same generic operation.

As opposed to previous attempts of using Concept Analysis for organizing data, we do not propose to navigate directly in the concept lattice. Instead, we use the contextualized logic (i.e., the logical view of the concept lattice) to evaluate the relevance of navigation links. Those that do not narrow the focus of the search are called *views*. They only restrict the language of available navigation links. Other links, that do narrow the focus of the search, can be used to come closer to some place of interest. The definition of links can be generalized to encompass data-mining notions like necessary and sufficient conditions, and association rules.

The advantage of LIS is a great flexibility which comes from two factors:

1. the integration of operations that were exclusive in most systems,
2. the use of logic with Concept Analysis, which solves the name problem.

We have experimented it in various contexts: e.g., cook-books, bibliographical repository, software repository (search by keywords, and search by types), and simply a note-pad. Various logic components were used in these contexts: atoms, intervals, strings, propositional logic, type entailment, taxonomies (e.g., for ingredients). In all cases, a LIS goes beyond any a priori structure and permits many kinds of views on the same information. For instance, in the case of a cook-books, if every recipe is described by its ingredients, its process, the required kitchen utensils, its dietetic value, its place in a meal, and more cultural information, then a cook, a dietician, and a gourmet can have very different views on the same data, and acquire new information by data-mining and learning, simply by using a few LIS shell commands. Similarly, if software components have intrinsic descriptions like their types and languages, the modules they use, parts of specification, and requirements, and extrinsic descriptions like their testing status, and who is using them, then several software engineering operations like developing and testing, versioning and configuring, and maintenance and evolution can be done using the same repository under different views, and also going smoothly from one view to another.

7.1 Related Works

There have been several other proposal of navigation/querying based on Concept Analysis. Lindig (1995) designed a concept-based component retrieval based on sets of *significant keywords* which are equivalent to our links for the logic of attributes underlying FCA. Godin et al. (1993) propose a direct navigation in the lattice of concepts, which is in fact very similar to Lindig's approach except that only greatest significant keywords, according to the contextualized subsumption on attributes, are displayed to the user. They have also notions common to our LIS such as working query, direct query specification, and history of selected queries.

Cole and Stumme (2000) developed a Conceptual E-mail Manager (CEM) where the navigation is based on Conceptual Scales (Prediger, 1997; Prediger and Stumme, 1999). These scales are similar to our views in the sense that they select some attributes acting as links and displayed, as for us, with the size of the concept they select. A difference with our LIS is that these links are ordered according to concept lattices of scales, but it can also be done in LIS by a post-treatment on answers of command `ls`.

However, the main difference with all of these approaches is that we use an (almost) arbitrary logic to express properties. This enables us to have automatic subsumption relations (e.g., (`author is "Wille, Mineau"`) $\sqsubseteq$ (`author contains "Wille"`) $\sqsubseteq$ (`author`)), and thus some implicit views (e.g., `author, year`). Another

difference is that we propose to handle in a uniform way, based on CA, navigation and querying as above, but also, updating, data-mining, learning, etc.

7.2 Future Work

Our most practical perspective is to design a *logical file system*, which would implement the ideas we have presented in this article, and serve as a Logic Concept Repository for a LIS. The expected advantage is to offer the services described here at a standard system level that is accessible for every application. So doing, even applications that do not know about logical information systems (like e.g., all existing compilers) would benefit from it.

A graphical user-interface to logical file systems would allow to display in an integrated fashion the working query, the working view, and the corresponding extent and set of links. For instance, a graphical interface for keeping trace of navigation, like what is becoming standard for file browsers, has been already experimented for a simple logic (attributes with values) but should be developed further. This amounts to keep a trace of the path from the start of the navigation to the current place. Moreover, the set of links could be presented graphically as a diagram of ordered formulas. A further refinement is to take into account the contextualized subsumption, to get something similar to concept lattices derived from scales (Cole and Stumme, 2000). This amounts to represent an overview of possible future navigations. However, it differs from usual concept analysis visualization tools (e.g., Vogt and Wille, 1994) in that it is not the concept lattice that is to be displayed.

The *World Wide Web* can also be explored using our techniques if one considers answers to web-queries as a formal context into which to navigate.

An application of the learning schema exposed in Section 5.2.1 is to use the learning schema for navigating. A description would play the role of a query, and its associative concepts could be proposed to the user as alternative queries. The advantage is that though the initial query could have an empty answer, the alternative ones always correspond to non-empty concepts. So, it makes it possible to start a search with only an example of what the user is looking for.

Another interesting perspective follows the observation that the notion of *associative concepts* is closely related to *modified* and *new concepts* in the incremental concept formation (Godin, Missaoui, and Alaoui, 1995). We plan to develop further this correspondence, since it may lead to improved algorithms for computing concepts incrementally.

In all this article, names or descriptions are essentially unary predicates, whichever is the actual logic used for this purpose. However, several applications require to express relations between objects, i.e., n-ary predicates. For instance, a LIS for a software environment should permit to express such relations as `calls` f or `is connected to` x, where f and x are objects. These relations form concrete links between objects, which we plan to consider for navigation in a future work. The main difficulty is to manage the concrete links in a way that remains compatible with the other navigation links. This will also permit to represent topological informations, e.g., `West of` x or 10 `miles from` y, that are used in Geographical Information Systems.

References

Chaudron, L., Maille, N., 1998. 1st order logic formal concept analysis: from logic programming to theory. Computer and Information Science 13 (3).

Cole, R., Stumme, G., 2000. CEM - a conceptual email manager. In: Mineau, G., Ganter, B. (Eds.), Int. Conf. Conceptual Structures. LNCS 1867. Springer, pp. 438–452.

Davey, B. A., Priestley, H. A., 1990. Introduction to Lattices and Order. Cambridge University Press.

Di Cosmo, R., 1995. Isomorphisms of Types: from λ-calculus to information retrieval and language design. Progress in theoretical computer science. Birkhäuser.

Ferré, S., 2001. Complete and incomplete knowledge in logical information systems. In: Benferhat, S., Besnard, P. (Eds.), Symbolic and Quantitative Approaches to Reasoning with Uncertainty. LNCS 2143. Springer, pp. 782–791.

Ferré, S., Ridoux, O., Dec. 1999. Une généralisation logique de l'analyse de concepts formels. Technical Report RR-3820, Inria, Institut National de Recherche en Informatique et en Automatique.

Ferré, S., Ridoux, O., 2000. A logical generalization of formal concept analysis. In: Mineau, G., Ganter, B. (Eds.), Int. Conf. Conceptual Structures. LNCS 1867. Springer, pp. 371–384.

Ferré, S., Ridoux, O., 2001. A framework for developing embeddable customized logics. In: LOPSTR. LNCS 2372. Springer, pp. 191–215.

Ferré, S., Ridoux, O., 2002. The use of associative concepts in the incremental building of a logical context. In: U. Priss, D. Corbett, G. A. (Ed.), Int. Conf. Conceptual Structures. LNCS 2393. Springer, pp. 299–313.

Ganter, B., Kuznetsov, S., 2001. Pattern structures and their projections. In: Delugach, H. S., Stumme, G. (Eds.), Int. Conf. Conceptual Structures. LNCS 2120. Springer, pp. 129–142.

Ganter, B., Wille, R., 1999. Formal Concept Analysis — Mathematical Foundations. Springer.

Gifford, D. K., Jouvelot, P., Sheldon, M. A., O'Toole, J. W. J., 1991. Semantic file systems. In: 13th ACM Symposium on Operating Systems Principles. ACM SIGOPS, pp. 16–25.

Godin, R., Missaoui, R., Alaoui, H., 1995. Incremental concept formation algorithms based on Galois (concept) lattices. Computational Intelligence 11 (2), 246–267.

Godin, R., Missaoui, R., April, A., 1993. Experimental comparison of navigation in a Galois lattice with conventional information retrieval methods. International Journal of Man-Machine Studies 38 (5), 747–767.

Gopal, B., Manber, U., 1999. Integrating content-based access mechanisms with hierarchical file systems. In: third symposium on Operating Systems Design and Implementation. USENIX Association, pp. 265–278.

Herzig, A., Rifi, O., 1999. Propositional belief update and minimal change. Artificial Intelligence 115 (1), 107–138.

Keller, A., 1985. Algorithms for translating view updates into database updates for views involving selections, projections, and joins. In: 4tf ACM Symp. Principles of Database Systems. pp. 154–163.

Krone, M., Snelting, G., May 1994. On the inference of configuration structures from source code. In: Int. Conf. Software Engineering. IEEE Computer Society Press, pp. 49–58.

Kuznetsov, S., 1999. Learning of simple conceptual graphs from positive and negative examples. In: Żytkow, J. M., Rauch, J. (Eds.), Principles of Data Mining and Knowledge Discovery. LNAI 1704. Springer, pp. 384–391.

Kuznetsov, S., Objedkov, S., 2001. Comparing performance of algorithms for generating concept lattice. In: et al., E. M. N. (Ed.), ICCS-2001 Int. Workshop on Concept Lattices-based Theory, Methods and Tools for Knowledge Discovery in Databases. Stanford University, CRIL – IUT de Lens, France.

Levesque, H., Mar. 1990. All I know: a study in autoepistemic logic. Artificial Intelligence 42 (2).

Lindig, C., 1995. Concept-based component retrieval. In: IJCAI95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs.

Prediger, S., 1997. Logical scaling in formal concept analysis. LNCS 1257 , 332–341.

Prediger, S., Stumme, G., 1999. Theory-driven logical scaling. In: International Workshop on Description Logics. Vol. 22. Sweden.

Snelting, G., Jul. 1998. Concept analysis — A new framework for program understanding. ACM SIGPLAN Notices 33 (7), 1–10.

Ullman, J., 1988. Principles of Database and Knowledge-Base Systems — Volume I. Computer Science Press.

Vogt, F., Wille, R., 1994. TOSCANA — a graphical tool for analyzing and exploring data. LNCS 894. pp. 226–233.

Wille, R., 1982. Ordered Sets. Reidel, Dordrecht Boston, Ch. Restructuring lattice theory: an approach based on hierarchies of concepts, pp. 445–470.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399