Software Engineering Applications of Logic File System

Application to Automated Multi-Criteria Indexation of Software Components

Benjamin Sigonneau
IRISA/Université de Rennes 1
Rennes, France
benjamin.sigonneau@irisa.fr

Olivier Ridoux
IRISA/Université de Rennes 1
Rennes, France
ridoux@irisa.fr

ABSTRACT

Logic information systems use formal concept analysis in a novel way to manage information. A file system implementation has been designed under the name of Logic file system. It offers a flexible management of non-hierarchical data. We present several applications of Logic file system to software engineering: multi-criteria indexation of software components, multi-concern browsing of source files, and bug finding in test traces.

We detail multi-criteria indexing of software components. Three independent indexing frameworks are developed and merged in a single multi-criteria framework. The three indexing frameworks capture formal criteria like type isomorphisms and inheritance relations, semi-formal criteria like naming conventions, and informal criteria like keywords of comments. We show how the logical orientation of Logic file system helps in capturing all these criteria in a single framework.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environment

General Terms

Design

Keywords

Logic information system, Software components

1. INTRODUCTION

Software engineering manages many kinds of documents that are related in many ways; it is the place for cross-cutting concerns and multi-view schemata. However, this rich network of relationships is often flattened on a single hierarchy. This is sometimes called the "tyranny of the dominant decomposition" [19]:

- The Java class browser is organized after the inheritance hierarchy; this makes it nearly impossible to search a method using different criteria (e.g. search for a method that returns a string, or search for a method that raises a particular exception).
- A standard source file displays the linear hierarchy of a text. One cannot claim to organize a source file using several criteria at the same time. One criterium must dominate the others.
- Object orientation brings types to the top of the hierarchy, so examining or creating a type is easy. At the opposite extreme, procedure orientation brings routines to the top of the hierarchy, so that examining or creating what concerns a type is scattered across source files, but examining or creating routines is easy.
- Sometimes, a hierarchy is imposed by a programming language. See for instance the layout of Java packages on directories.
- Even when it is electronic and enriched with navigation links, documentation is often thought as a rigid document, in which an almost arbitrary priority of concerns concentrates favoured concerns in a few sections, and discards unfavoured concerns across the whole document.

In all these situations, finding a particular piece of information requires either luck or erudition. We contend that neither luck nor erudition is a necessary practice for robust software engineering.

We feel that the very heart of this problem lies in the fact that traditionnal information systems (e.g. hierachies or databases) fail to organize this data in an efficient way that would be convenient to the user, and artificially enforce the use of a dominant decomposition. We propose to use a new powerful organization framework, called *Logic Information System* (LIS), to manage software engineering artifacts.

In Section 2 we will present what a logic information system is and a file system implementation of a LIS. Then we will present in Section 3 the outlines of three experiments of software engineering applications based on the LIS file system. Finally, we will present in more details the retrieval of Java methods using the LIS file system (see Section 4).

2. LOGIC INFORMATION SYSTEMS

Hierarchical organizations are rigid because only one path leads to every single object. Sometimes, this rigidity is relaxed by introducing links, but this does not scale well because navigation does not work well with links, and generally nothing prevents from creating dangling links. In short, links are an afterthought[1]. The good news about hierarchical organization is that it makes a notion of place very intuitive (every node in the hierarchy is a place), and it

[1]Even in cases where links are not an afterthought, e.g. in a web site, the same rigidity arises since only a *fixed* set of paths leads to every object.

MSR'06, May 22–23, 2006, Shanghai, China.

makes navigation progressive. Indeed, if we consider the hierarchy of a UNIX file system, the answer to an `ls` in a place (e.g. a directory) is the set of objects that inhabit the place, and a set of sub-places (e.g. subdirectories). It is only when forced that a hierarchical organization dumps all its content (say, using `ls -R` in a UNIX file system).

A truly different organization is the boolean organization, where objects are associated with attributes, and the answer to a query is the set of objects that are associated to some attributes, as can be seen in Google. It is very flexible because attributes can be queried in any order and any combination. However, it is not progressive at all because it dumps all objects that answer the query without organizing them, and it gives no hints on how the query could be refined. So, there is no notion of place, hence no real navigation.

The goal of logic information systems (LIS, for short) is to get the best of both worlds: flexibility in queries, a clear notion of place, and navigation.

2.1 Formal Framework

This section presents the outline of the theory of LIS. Full details can be found in [4].

The contents of a LIS is a set of objects, O, and a mapping d (for *description*) from objects to properties. Objects can be files, like photos and source files, or parts of files, like procedures in a source file. Properties are expressed as formulas of some logic $\mathcal{L}$. Very little is demanded on the logic; its entailment relation (written $\models$ in the sequel) must handle a form of conjunction and disjunction (written $\wedge$ and $\vee$), have a tautology (written $\top$), and be monotonic and decidable. Since the LIS theory was inspired by *formal concept analysis* (FCA [6]), the *object* $\mapsto$ *property* mapping is called a *context*. The logic $\mathcal{L}$ used for expressing properties is a generic parameter of the theory. In practice, well-known logics like proposition logic are not used alone; very specific logics, e.g. for comparing dates, are much more useful. A logic of containment of sets of keywords is often good enough. In this case, LIS comes very close to standard FCA. In standard FCA a context may be infinite, but in LIS it will always be finite.

For convenience, we will say that "an object entails a property p" if its description entails p. The theory of FCA defines the intention of a set of objects as the most specific property that describes *them all*. Dually, the extension of a property is the set of *all* objects that entail the property.

Definition 1 (Intention and extension).

$$\begin{array}{lll} int(O) & = \bigvee_{o \in O} d(o) & \text{where } O \subseteq \mathcal{O} \\ ext(p) & = \{o \in \mathcal{O} \mid d(o) \models p\} & \text{where } p \in \mathcal{L} \end{array}$$

Clearly, intention and extension map set of objects to properties and back.

In fact, $ext \circ int$ and $int \circ ext$ are called *closures*; they are normalizing operators. Note that the normal form of a set of objects or of a property depends on the context. In general, not every subset of O is normal, and there are much less normal subsets than there are subsets. Similarly, not every property is normal.

This shows that pairs $\langle ext(p), int(ext(p))\rangle$ and $\langle ext(int(O)), int(O)\rangle$ are special. They are extension-intention pairs in which the intention is the intention of the extension, and the extension is the extension of the intention. FCA theory calls these pairs *concepts*. Concepts can be ordered: a concept is smaller than another if its extension is contained in the other, or equivalently if its intention entails the other. In LIS, there are always finitely many concepts.

Definition 2 (concept). *A pair $\langle e, i\rangle$ is a* concept *iff $ext(i) = e$ and $int(e) = i$. A concept $\langle e, i\rangle$ is smaller than or equal to a concept $\langle e', i'\rangle$, $\langle e, i\rangle \leq \langle e', i'\rangle$, iff $e \subseteq e'$ (or equivalently $i \models i'$).*

The main result of FCA is that given a context, the set of all its concepts ordered by $\leq$ forms a complete lattice: the *concept lattice*. Furthermore, the original context can be reconstructed from the concept lattice. In some sense, the context is the concrete data-structure, and the concept lattice is the information it contains. In LIS words, the contents of a LIS is a context, but LIS shows concepts as places, and the $\leq$ relation as a subdirectory relation.

$\langle ext(int(\{o\})), int(\{o\})\rangle$ is always a concept, but the converse is not true, simply because there may be more concepts than objects. This concept, written $conceptof(o)$ in the sequel, is said to be *labelled by o*. Similarly, $\langle ext(p), int(ext(p))\rangle$ is always a concept, but it depends on the logic whether there are more concepts than formulas (modulo entailment). It is written $conceptof(p)$, and we say that p labels this concept. The practical interest of this notion is that $\{o\}$ is usually much smaller than $ext(int(o))$, and p is also usually much simpler that $int(ext(p))$.

Finally, one says that properties that label the same concept are *contextually equivalent*. This shows three levels of information in a LIS:

1. Absolute truth is represented as the logic of properties. At this level, $a \wedge b \models a$ or *fish* $\models$ *vertebrate* whatever happens in the context. It is up to the administrator of a LIS to choose what absolute truth he wants to represent.
2. Facts are represented in the context. New facts may be added, old facts may be deleted.
3. Concepts, and contextual entailment, represent contingent truth, i.e. conclusions that may not be valid in another context.

In his logic modeling of information systems querying, Van Rijsbergen [20] proposed that the answer to a query be its extension. However, an extension may be large, especially if the query is vague. So, LIS takes another stand-point [4]:

Definition 3 (conceptual navigation). *The answer to a query q is $dirs(q) \cup files(q)$ where:*

$$\begin{array}{ll} dirs(q) & = \text{a finite set } P \text{ such that} \\ & \quad \forall c < conceptof(q). \left[\begin{array}{l} \neg \exists c' < conceptof(q). c < c' \\ \Rightarrow \exists p \in P. conceptof(q \wedge p) = c \end{array} \right. \\ files(q) & = \{o \mid conceptof(o) = conceptof(q)\} \end{array}$$

The $files(q)$ are the objects at place q, i.e. the files in directory q, whereas the $dirs(q)$ are properties that reach the greatest lower concepts, i.e. the subdirectories of q. Elements of $dirs(q)$ are also called *increments* to avoid the connotation of a file system, and to reflect their use in incremental navigation.

Note that the answer to a query contains other queries (see $dirs(q)$). This is analogous to a dialog between a customer (C) and a shop assistant (SA):

C: I want to buy flowers! What do you have?
SA: Do you have any idea of the color, kind of flower or size of bouquet?
C: I want a big bouquet! What color do you have?
SA: Red, white or yellow.
...

The user never has to guess a formula; he only has to select a formula among the $dirs(q)$ and keep on repeating the process until he finds the object he was looking for. So, a user may even navigate in a context with a logic he does not know, provided he understands the formula. In other words, a passive knowledge of the logic is enough. However, if he has a deeper knowledge of the logic, he can go faster by setting his initial query q directly to a formula that caracterizes the desired object.

To sum it up, a LIS behaves like a schemaless database, its organization structure being computed dynamically from the logical descriptions of the objects. In a database system as in a LIS, queries are intentional. The most striking difference between a database system and a LIS is the nature of the answer. Whereas it is *extensional* in a database system, i.e. objects, in a LIS the answer to a query is also *intentional*, i.e. expressed in the same language as queries. Therefore, the search can be progressive through an iteration process: ask for a query, pick up an answer to complement the query, etc.

2.2 A LIS File System — LISFS

LIS is a candidate for replacing the traditional hierarchical organization when it does not fit well the needs of an application. In fact, a hierarchical organization is often used as a default solution, despite its lack of flexibility. Many applications contain their own browser in order to circumvent the inadequacy of an underlying hierarchical store. Having a LIS as a file system would provide an adequate solution that is ready for use by different applications. As a bonus, this would help making these applications communicate through the file system.

So, we have studied an implementation of LIS as a file system in order to offer a generic service that could be used in already existing applications [13]. This implementation, named LISFS, uses as much as possible database and file system techniques. It is still a prototype but its current state shows acceptable performance for interactive usage with more than 100,000 files.

In modern operating systems, a file system is an implementation of some file system interface (e.g. VFS for Linux, *virtual file system*). Although a file system is best presented at this level, this would reveal technical details that are irrelevant in this article. So, we choose to present LISFS at the shell level. We have not written a new shell; we simply run an existing shell on LISFS.

In LISFS, a fragment of propositional logic with valued attributes is proposed as a kernel logic. To achieve this, paths are considered as formulas; a path is a conjunction of atomic properties (directories)[2], i.e. the UNIX path separator `/` is to be read as a logic conjunction $\wedge$. This gives a new semantics to old shell commands.

EXAMPLE 1 (USING LISFS).

```
[1] mkdir a; mkdir b; mkdir c; mkdir d
[2] touch a/b/d/fabd; touch d/c/b/fbcd; touch d/b/fbd
[3] cd b; ls
    a/ c/ fbd
[4] cd a; ls
    fabd
```

Under LISFS, the `mkdir` *command creates axioms. So at Line 1, four directories are created at the root (*`pwd` $= \top$*), i.e. axioms* `a` $\models \top$*, etc. are declared. Then, three files are created at Line 2 with the standard UNIX command* `touch`*:* `fabd` *has property* `a` $\wedge$ `b` $\wedge$ `d`*,* `fbcd` *has property* `b` $\wedge$ `c` $\wedge$ `d`*, and* `fbd` *has property* `b` $\wedge$ `d`*.*

Starting from the root, command `cd` *is used to go into subdirectory* `b` *at Line 3. Adopting the LIS point of view, this reads: property* `b` *is selected.* `pwd` *becomes* $\top \wedge$ `b` $=$ `b`*.*

This directory contains two subdirectories, `a/` *and* `c/`*, and a file* `fbd`*. Though* `b` $\wedge$ `d` *is not equivalent to* `b`*, it is contextually equivalent. At Line 4, property* `a/` *is selected, setting* `pwd` *to* `a` $\wedge$ `b`*. Only one file remains, namely* `fabd`*.*

Moreover, LISFS can be extended by using a *plugin* mechanism. In this way, LISFS maintains the genericity of the design of LIS. This mechanism also handles to kinds of plugins : *logics* are used to attach new specific logics to some properties (e.g. interval logics or date logics) and *transducers* are programs that can extract properties from file contents as soon as a file is updated. So, the description of a file can be made of both *intrinsic* properties computed and maintained by transducers, and *extrinsic* properties maintained by the user.

Another very important feature of LISFS is that it may operate at two levels. At the first level, called *interfile*, objects are files. At the second level, called *intrafile*, objects are parts of files [14]. The first level is the standard operation level of file systems. The second level permits to treat a file as a directory and explore its constituents. The extension of the `pwd` is always a selection of parts of the original file. In every directory, it is presented as the unique inhabitant of directory `pwd`. At the intra-file level, the extension can be considered as a *slice* or a *view* of the original file for some property. Moreover, these *views* can be edited; the modifications will be retropropagated to the original file, and to every other view that shares something with this one. This unifies access methods inter- and intrafiles, and it makes tools designed to browse a collection of files suitable to browse the components of a file.

2.3 Applications

We have experimented with applications ranging from personal information systems for managing recipes, agenda, music files, photos, one's homedir, to software engineering tools and office applications for managing bibliographies and emails. We are also currently developing a geographic information system prototype.

3. SOFTWARE APPLICATIONS OF LIS

3.1 Indexing Software Components

Component retrieval is a key issue for the hability to reuse components. Prieto-Díaz explains that this implies classifying components, and that it can be done in two ways: hierarchical or faceted classification [15]. He further explains that faceted classifications are more suitable for classifying software components, though they require the intervention of an expert, and they do not work well for heterogenous collections. The requirement for an expert can be avoided if facets are extracted automatically. Furthermore, LIS are designed for coping with heterogenous data. So, we propose to use LISFS as a storage device for a component manager.

Designing such an application amounts to designing a logic for representing component properties, and the structure of the context. The attachment of a property to every single component is automated by a transducer, and LISFS supports the run-time retrieval system. This approach can be transposed to any kind of software components like Web services [12, 9] or COTS [17]. Section 4 presents an application of this approach to Java methods.

3.2 Browsing Source Trees and Source Files

LISFS can be used to browse source trees and sources files.

Regarding source trees, LISFS has been used to manage the Linux kernel source tree. In this case, intrinsic properties of the files are the names of the functions they contain (as extracted with the ctags command). The files also contain extrinsic properties that correspond to their path in the original source tree (e.g. `driver`, `fs`, `include`, etc.).

With such a huge source tree, one would like to classify the files according to his current needs: by file type (includes, C files, text documentation, ...), by architecture (`x86`, `PPC`, ...), by module (drivers, virtual memory manager, ...) and so on. However, the

[2]In fact, a path is a conjunction of formulas that can be atomic, disjunctive (e.g. `small|large`), conjunctive (e.g. `black&blue`), or negative (e.g. `!costly`). The symbol `&` is only necessary for conjunctions that are nested in disjunctions or negations

organization of the Linux kernel source tree suffers from the limitations of hierarchical file systems and therefore forces the developer into using a fixed organization scheme (currently, by type, then by module, then by architecture). Some directories therefore appear in different places (e.g. there are 4 `sound` directories), the organization is not always coherent and is hard to understand. One answer to these problems is to use specific tools, such as LXR [10], a cross-referencing tool for relatively large code repositories.

Another solution is to use LISFS, with which these problems become irrelevant. The classification scheme is not fixed anymore, and the user can select the set of files dealing with memory handling, should they be includes, C files or documentation with `cd mm`. Or he could have retrieved every include files with the same ease: `cd kind:header`, focused on a particular architecture: `cd architecture:ppc`. Of course, those queries can be issued in any order, and in any place ranging from the root of the source tree to the deepest subdirectory where it is still relevant.

Once a source file is located in the tree, the programmer is able to work on it. However, a source code contains scattered information, such as debugging statements, comments, assertions, etc. As recognized by Aspect-Oriented Programming [8], this cross-cutting information is generally not handled in an easy way because it is scattered in the program. With LISFS intra-file mode, it becomes easy to manage such cross-cutting concerns.

To do so, the user has to input the special query `cd parts`; then, the file is diplayed along with several subdirectories, each of them leading to a partial view of the initial file. For this to work under LISFS, plugins need to be designed, one for each kind of programming languages. Such plugins have been developed for several languages, among which C source files (used when working on the Linux kernel sources) and OCaml source files (used when working on LISFS source code).

Therefore, we are able to ask for a partial view of source files written in those languages. E.g., we can ask for a view of a source file with comments and without debugging statements by issuing the query `cd comment|(!aspect:debug)`. The result is a place that holds a copy of the file where the non-selected parts (here, the debug statements) are hidden. Any change operated on this partial view will be propagated in the original file. Moreover, the result of an `ls` command shows other subdirectories, i.e. it prints out how to get smaller views of the source file. In the case of C files, it shows we can focus on a particular `function:`, or make a slice regarding a particular variable (`var:`), and so on.

3.3 Analysis of Program Traces

LISFS can be used to explore execution traces of programs.

In a first approach, we focus on Prolog program execution traces. Every event of a trace is described by a text line that mentions the values of a set of attributes (event number, port, predicate, goal, depth) reflecting Byrd's box model [1]. Using LISFS intrafile level, a trace can form a context where events are objects to be browsed. For instance, `cd depth:>5` gives a partial view of the trace showing only goals whose depth is greater than 5.

Another approach is to query a pool of different execution traces of the same program corresponding to different test cases, or of different execution traces of different mutants on the same test case. In this case, traces are the objects of the context. They are described by test verdicts (*pass* or *fail*), and the numbers of the executed lines, or other trace information. The problem of locating bugs using traces and test verdicts is not new [7], but it was recently rephrased in the context of data-mining [2]. The advantage is that data-mining them gives well-known indicators (e.g. support, confidence, lift) that can be used in a more principled way than the *ad hoc* indicators used by Jones *et al.*

We propose to use LISFS to crosscheck the traces so as to locate errors in the code. A further advantage of using LIS instead of the standard association rules method is that LIS can accommodate a large range of logical descriptions whereas association rules are limited to attribute-based contexts. So, we expect to be able to use other trace descriptors than merely line numbers. Preliminary experiments are currently being lead on C programs execution traces.

4. SOFTWARE COMPONENT INDEXING

We present more details on the application for indexing software components. In this section, components are Java methods.

4.1 A Logic for Classifying Java Methods

We classify facets as *formal*, *semi-formal*, and *informal*. We did not try to implement every possible facet, but we decided to implement one in each kind of facets. This shows how very different kinds of descriptions combine in a single property, and are used in navigation.

Formal facets are formally related to the semantics of components; we have chosen types because they are already given in the source, though any static property would do. Concerning type, we have developed an entailment relation which combines type isomorphisms [3] with the inheritance relation. This is a contribution in itself since prior attempts to do so in the higher-order type case lead to a contradiction [18], while we show that the first-order type case works well. From a LIS perspective this also shows how very specific logics can be developed to fit a given purpose.

Semi-formal facets are only loosely connected to the semantics; we have chosen methods identifiers, and especially the convention that helps splitting an identifier into a phrase. For instance, `getValue` is usually meant to be read as "get value". This reading is not connected to the semantics because nothing forces a method `getValue` to get anything, and *vice versa*. However, identifiers are formally connected to the semantics by the store; a spelling mistake is seen by the compiler. In a given context, whatever `getValue` means is formally defined.

On the opposite, we also developed an informal facet based on comments; they are not formally connected to the semantics at all, and a spelling mistake is not seen by the compiler.

We propose a type entailment relation that is based on inheritance, written $t \leq_{\text{inh}} t'$ iff t inherits from t', and on type isomorphism, written $t \sim_T t'$ iff t is isomorphic to t' wrt. theory T. We consider Java as a first-order object-oriented language in which the only polymorphism comes from inheritance; there is no polymorphism *à la* ML. The language of Java types is abstracted as follows:

DEFINITION 4 (TYPES).

$$\begin{array}{lcl} \mathit{Type} & ::= & \mathit{Arg} \to \mathit{Res} \\ \mathit{Arg} & ::= & \mathit{Arg} \times \mathit{Arg} \mid \mathit{class} \\ \mathit{Res} & ::= & \mathit{class} \end{array}$$

We write $x : t$ iff x has type t.

In order to use types as descriptions of objects in LIS, hence as queries, we have to define what is the entailment relation of types considered as a logic. First, we show intuitively that the arrow type must be *contravariant* for the entailment relation.

DEFINITION 5 (CONTRAVARIANCE). *Let $\leq$ be a partial order defined on types as in Definition 4. Relation $\leq$ is said to be contravariant iff*

$$\forall \sigma, \sigma', \tau, \tau', \sigma' \leq \sigma, \tau \leq \tau' \quad \textit{implies} \quad \sigma \to \tau \leq \sigma' \to \tau'$$

Indeed, any entailment relation can be considered as a partial order. Moreover, type entailment must extend the inheritance relation, i.e. $t \leq_{\text{inh}} t' \Longrightarrow t \models t'$, because it is interpreted as $t \leq_{\text{inh}} t' \Longrightarrow ext(t) \subseteq ext(t')$. For instance, `Number` $\leq_{\text{inh}}$ `Object` means that every `Number` is an `Object`, but the opposite is false. Finally, the arrow type must be considered as contravariant for the inheritance relation, hence it must be considered as contravariant for the entailment relation too because the latter extends the former. Why must the arrow type be considered as contravariant for the inheritance relation? Let us take an example. Assume a user is looking for a method that takes a `Button` as a parameter and returns a `Container`. His query will be `Button` $\rightarrow$ `Container`. Every method that accepts a super-type of `Button`, plus supplementary parameters, and that returns a subtype of `Container` is a correct answer. This shows that parameters and results play opposite roles. In fact, type entailment means "can replace the other, and still be type-checked".

When taken literally, types are not good search keys because two types may differ though semantically neutral transformations would make them identical. For instance, the order of parameters should not matter because a parameter permutation makes types identical. This is called *type isomorphism*, and has been recognized for long as the key to use types as queries for searching software components [16, 18].

Definition 6 (Type isomorphism). *A type t is isomorphic to a type t', written $t \sim t'$, iff*

$$\exists f : t \rightarrow t' \quad \exists g : t' \rightarrow t \quad [g \circ f = \mathrm{Id}_t \wedge f \circ g = \mathrm{Id}_{t'}]$$

Functions f and g are called the witnesses *of the isomorphism. They are the semantically neutral operations that make two types equivalent.*

It is convenient to present type isomorphisms as equivalence relations wrt. a set of axioms. Every set of axioms generates an isomorphism. For instance, axiom $\sim_{\text{exch}}$ (exchange) says that $t \times t'$ and $t' \times t$ must be considered as isomorphic, and axiom $\sim_{\text{curry}}$ (curryfication) says that $t \rightarrow (u \rightarrow v)$ and $(t \times u) \rightarrow v$ are isomorphic. Isomorphism axioms generate equivalence classes that make a type the representent of each type of its class. Di Cosmo has developed a complete theory of isomorphism axioms [3], however $\sim_{\text{exch}}$ is the only axiom that is relevant to our type language. Other axioms, like $\sim_{\text{curry}}$, deal with characteristics of other type systems like higher-order.

The entailment relation can also be presented as a set of axioms. We already know the $\models_{\text{inh}}$ axiom, which says that entailment extends inheritance. A second axiom is $\models_{\text{drop}}$, which says that $(t \times u) \rightarrow v \models t \rightarrow v$. Other axioms have been developed in the literature, but they do not apply to our type language; e.g., $\models_{\text{inst}}$ which says $\forall \alpha.t \models t[\alpha \leftarrow t']$ applies to polymorphism *à la* ML.

Finally, an entailment relation can be built by combining isomorphism axioms, hence considering equivalent classes of types, and entailment axioms. However, entailment axioms may generate new equivalence classes, e.g. if $t \models t'$ and $t' \models t$. It may even make the entire set of types collapse into too few equivalent classes. This is what happens when isomorphism axiom $\sim_{\text{curry}}$ and entailment axioms $\models_{\text{drop}}$ and $\models_{\text{inst}}$ are combined [18]. So, it is important to study what happens with axioms that apply to our type language: $\sim_{\text{exch}}$, $\models_{\text{drop}}$ and $\models_{\text{inh}}$. In fact, we have proven that every equivalence class induced by $\sim_{\text{exch}}$, $\models_{\text{drop}}$ and $\models_{\text{inh}}$ is already induced by $\sim_{\text{exch}}$.

Theorem 1 (Soundness of $\sim_{\text{exch}}$,$\models_{\text{drop}}$ and $\models_{\text{inh}}$).

$$\forall t, t' \left[\begin{array}{l} \left[\begin{array}{c} t \models t' \text{ wrt. } \sim_{\text{exch}},\ \models_{\text{drop}}, \text{ and } \models_{\text{inh}} \\ \wedge \\ t' \models t \text{ wrt. } \sim_{\text{exch}},\ \models_{\text{drop}}, \text{ and } \models_{\text{inh}} \end{array} \right] \\ \Longrightarrow t \sim t' \text{ wrt. } \sim_{\text{exch}} \end{array} \right]$$

To conclude, we choose as a logic for representing types the entailment relation induced by axioms $\sim_{\text{exch}}$, $\models_{\text{drop}}$ and $\models_{\text{inh}}$.

The entailment relation for keywords is much simpler. Semi-formal and informal properties are sets of keywords, and we say that $s \models s'$ iff $s \supset s'$.

4.2 Implementation

The previous section has shown the logical engineering one must engage into to develop a formal method. It is not related to LISFS. This section will show how LISFS helps in implementing the resulting entailment relation at almost no cost.

We could have developed a logic plugin for type entailment, but LISFS allows an alternative solution. The native solver of LISFS already implements a fragment of propositional logic that contains the exchange rule, $\frac{A \wedge B}{B \wedge A}$, and the weakening rule, $\frac{A \wedge B}{A}$. These rules correspond to axioms $\sim_{\text{exch}}$ and $\models_{\text{drop}}$ at the propositional level. So, we propose to represent types in such a way that exchange and weakening will implement axioms $\sim_{\text{exch}}$ and $\models_{\text{drop}}$.

Contravariance tells that types behave differently according to their context. We call *positive* the types of the right end of $\rightarrow$, and *negative* the types of the left end. Similarly, exception types are called positive, because they behave as results, and class types are called negative, because they behave as parameters. So, we represent complex types as conjunctions of signed base types.

Definition 7 (Encoding of types as conjunctions). *We note $\lceil . \rceil$ the encoding of types into conjunctions of signed base types. Remember that $--t = +t = t$.*

$$\begin{array}{lcl} \lceil t \rightarrow t' \rceil & = & -\lceil t \rceil \wedge \lceil t' \rceil \\ \lceil t \times t' \rceil & = & \lceil t \rceil \wedge \lceil t' \rceil \\ \lceil \textit{base type} \rceil & = & \textit{base type} \end{array}$$

Using encoding $\lceil . \rceil$, the implementation of axioms $\sim_{\text{exch}}$ and $\models_{\text{drop}}$ comes for free. What remains to implement is the $\models_{\text{inh}}$ axiom. It is simply done by using LISFS command `mkdir` which implements user-defined axioms (see Section 2.2). Every time a declaration says that a class `A` inherits from a class `B`, symbols `in-A` and `out-A` are created for representing $-$`A` and +`A`, and a command `mkdir out-B/out-A` creates the axiom +`A` $\models$ +`B`. Once all the inheritors of a class `B` are known, say $\mathtt{A}_1, \ldots, \mathtt{A}_n$, a command `mkdir in-`$\mathtt{A}_1$`/.../in-`$\mathtt{A}_n$`/in-B` creates the axioms $-$`B` $\models -\mathtt{A}_1$, $\ldots$, $-$`B` $\models -\mathtt{A}_n$.

All this implements the desired entailment relation without having to develop a theorem prover.

4.3 Experiments

We have experimented our Java method browser on existing Java packages. Consider, for instance, the AWT package. It consists of about 5,200 methods. The context builder passes through the package and creates its lines and columns in two passes: type first, then identifiers and comments. Figure 1 shows the shape of the actual AWT context. Indeed, every symbol has an internal identifier in LISFS (its inode), and the figure simply shows the context as a matrix. The figure can be interpreted as follows.

Objects, i.e. AWT methods, are on the horizontal axis, and attributes on the vertical one. Object numbers are introduced in the order of method declarations in the program. Types are analyzed before identifiers and comments, so they are given the low attribute numbers (below $\approx$1,700 on the figure). Types are numbered during a traversal of the inheritance graph. Once this is done, identifiers and comments are read in the source order, so they are given the highest attribute numbers. The figure therefore displays a type area, and a keyword area. The type area is strongly structured by lines

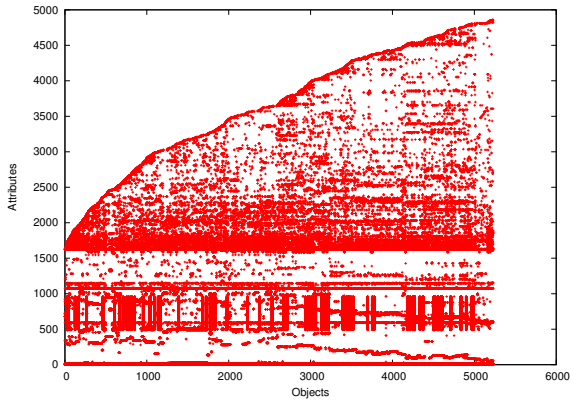

Figure 1: AWT formal context

and columns. Lines show types that are shared by many objects, whereas columns show types that come together in a single object. Columns are mainly an effect of inheritance. The keyword area is the triangular part. It shows that keywords that appear first are also frequent keywords since the triangle basis is darker than its top. The keyword area shows no particular structure. LIS and LISFS propose a rational navigation principle in this kind of structure.

Let us assume that a user looks for a method that takes a string as a parameter. The following simple commands help analyzing the situation.

```
[1] cd /mnt/lisfs/component-manager/
[2] cd in-java.lang.String
[3] ls | wc -l
        444
[4] ls .ext | wc -l
        356
```

We present the queries as shell commands, but one should rather imagine them as buttons of a graphical interface. The last two commands show that there are 356 methods that takes a string as a parameter, and that there at 444 ways to make the user demand more precise. So, the user had better think a little on his own, and he remembers that what he wants to do is related to MIME types.

```
[5] cd 'ident:mime|comment:mime'
[6] ls
     1 in-java.awt.datatransfer.MimeTypeParseException/
     2 in-java.awt.datatransfer.MimeTypeParameterList/
     3 static/
     3 in-java.awt.datatransfer.SystemFlavorMap/
     3 in-java.awt.datatransfer.DataFlavor/
     4 in-java.awt.datatransfer.MimeType/
     5 by-exception/
     7 out/
    11 comment/
    12 ident/
    12 method/
    12 by_class/
    12 access_control/
```

Of the 5,200 methods of the initial context, only 12 are possible answers, and there are few relevant increments. So, the user can study them and recall that the command he is looking for is not static.

```
[7] cd '!static'
[8] cd .ext
[9] ls
    MimeTypeParameterList       MimeTypeParseException
    isMimeTypeEqual             normalizeMimeType
    normalizeMimeTypeParameter  parse
[10] cat normalizeMimeType
    Called for each MIME type string to [...]
```

The user has recognized the name `normalizeMimeType` and checked its summary. Overall, the navigation took 3 steps that invoked the three kinds of properties. Each kind of property determines a classification of Java methods in itself, but it is LIS that combines them all in a single classification.

Former propositions by Rittri, Runciman, and Di Cosmo described classifications based on type isomorphism, but they lacked proper ways to navigate in possible answers. Furthermore, in their propositions the user is to submit a nearly complete expected type. In ours, he only submits the part that seems relevant.

Another possible classification of Java methods is by using the inheritance graph of the classes they belong to. This is the only classification used in the official Java documentation. Assume the developer of a graphical interface looks for a method that would return the name of a window. He may start by exploring the class `Window`.

```
[11] cd /mnt/lisfs/component-manager/class-java.awt.Window
[12] ls
      3  static/
     11 by_exception/
     24 final/
    255 class-java.awt.TextField/
    255 class-java.awt.TextArea/
    255 class-java.awt.Scrollbar/
    255 class-java.awt.List/
    255 class-java.awt.Label/
    255 class-java.awt.Choice/
    255 class-java.awt.Checkbox/
    255 class-java.awt.Canvas/
    255 class-java.awt.Button/
    328 comment/
    361 class-java.awt.ScrollPane/
    361 class-java.awt.Panel/
    361 class-java.awt.Container/
    438 ident/
    438 method/
    438 by_type/
    438 access_control/
[13] ls .ext | wc -l
       438
```

LISFS computes 20 increments in a few seconds. They concern 438 AWT methods. Since he is looking for window names, the developer searches for keyword “name”.

```
[14] cd ident:name
[15] ls
    1 ident:set/
    1 ident:get/
    2 ident:construct/
    2 ident:component/
    3 class-java.awt.TextField/
    3 class-java.awt.TextArea/
    3 class-java.awt.Scrollbar/
    3 class-java.awt.ScrollPane/
    3 class-java.awt.Panel/
    3 class-java.awt.List/
    3 class-java.awt.Label/
    3 class-java.awt.Container/
    3 class-java.awt.Choice/
    3 class-java.awt.Checkbox/
    3 class-java.awt.Canvas/
    3 class-java.awt.Button/
    4 comment/
    4 by_type/
    4 access_control/
```

Keyword “get” seems relevant. A quick look will confirm it.

```
[16] cd ident:get
[17] ls
     getName
[18] cat .ext/getName
    Gets the name of the component.
```

These experiments confirm the feasibility and interest of combining a wide range of properties. The user submits queries using properties that are relevant for him, and LIS return relevant increments. The navigation may start by submitting type information and continue using keywords; the user is not bound to using a single classification.

5. CONCLUSION

LIS and LISFS form a flexible and powerful framework for developing software engineering tools. We have already made experiments in source browsing, component browsing, and trace analysis for bug finding. As it includes basic logic services, LISFS can also help in the logical engineering of an application.

We have also presented the design of a component browsing tool that we have demonstrated on real Java packages.

In its current state, LISFS can only handle contexts in which properties are attached to one object at a time; in other words, properties are unary predicates. We are developing a variant of LIS in which properties can be attached to vectors of objects [5]. So doing, properties are n-ary predicates. This is especially important in software engineering applications because they are rich in inter-objects relations, e.g. calls, imports, compiles-to, inherits-from, etc. These relations can be simulated with unary predicates, but it is inconvenient. For instance, maintaining an inverse relation is error-prone.

The concept lattice is based on strict containment, so that navigation increments are always relevant. However, it could be useful to consider qualified containment like 90% of concept c belongs to concept c'. This leads to introducing *association rules* and data-mining operations in LIS. We believe it is useful in exploratory applications like fault-finding [2].

LIS and LISFS propose a navigation metaphor based on the notion of place through the use of increments; in particular the concept lattice is never actually built. Other authors have proposed to navigate graphically in the concept lattice. We believe this is impracticable because there are too many concepts, even in not so large contexts. For instance, the AWT context produces about 135,000 concepts. However, what LISFS computes is only a path through it.

Facetted classification *à la* Prieto-Díaz and type isomorphisms *à la* Di Cosmo are opposite methods for component finding. The former is informal and manual, while the latter is formal and automated. However, LISFS combines both approaches. We believe it is a step towards an answer to Mili *et al.*'s remark, "*no solution offers the right combination of efficiency, accuracy, user-friendliness and generality to afford us a breakthrough in the practice of software reuse*." [11]

generality: On the one hand, LIS as a framework is generic with respect to the logic used in object descriptions. There is no prerequisite on classifications, and on their number. On the other hand, LISFS as a file system integrates well with other tools. For instance, file browsers automatically become concept browsers when mounted on a LISFS partition.

user-friendliness: Because it is generic, LIS can adapt to the user's preferred logic. Furthermore, it proposes a dialog-based navigation in which the user needs only a passive knowledge of the description language. This makes it easy to grasp LIS progressively. As a file system, LISFS offers nothing directly but can be easily used through a graphical interface.

accuracy: As a formal framework, LIS behaviour is completely defined. In particular, navigation is complete in a formal sense (i.e. every object can be accessed using only increments returned by LIS; the user need not invent queries), and increments are always relevant (i.e. they never lead to dead-ends, and they are always focusing on a strict subconcept of the current concept). Combining classifications also increases accuracy because an application based on several classifications will always be at least as accurate as the most accurate of the classifications. So, if a classification is good for homogenous contexts and another is good for heterogenous contexts, the combination of both will be good in both cases.

efficiency: LISFS is an efficient implementation of LIS that can handle up to 100,000 objects with reasonable performance. This is still less efficient than an ordinary file system or database. However, it offers more services and it is still in its infancy; hierarchical file systems and databases have been improved by about 30 years of intensive usage. We believe LISFS can improve its peformance to the level of state-of-the-art file systems.

6. REFERENCES

[1] L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-Å. Tärnlund, editor, *Proc. of the Logic Programming Workshop*, Debrecen, 1980.

[2] T. Denmat, M. Ducassé, and O. Ridoux. Data mining and cross-checking of execution traces. A re-interpretation of Jones, Harrold and Stasko test information visualization. In T. Ellman and A. Zisman, editors, *20th Int. Conf. on Automated Software Engineering*. ACM Press, 2005.

[3] R. Di Cosmo. Deciding type isomorphisms in a type-assignment framework. *Journal of Functional Programming*, 3(4):485–525, 1993.

[4] S. Ferré and O. Ridoux. Introduction to logical information systems. *Information Processing and Management*, 40(3):383–419, 2004.

[5] S. Ferré, O. Ridoux, and B. Sigonneau. Arbitrary relations in formal concept analysis and logical information systems. In F. Dau, M.-L. Mugnier, and G. Stumme, editors, *ICCS*, volume 3596 of *LNCS*. Springer, 2005.

[6] B. Ganter and R. Wille. *Formal concept analysis — Mathematical Foundations*. Springer, 1999.

[7] J. Jones, M. J. Harrold and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Int. Conf. on Software Engineering*, 2002.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*, volume 1241 of *LNCS*. Springer-Verlag, 1997.

[9] J. Liberty. *Programming C#*. O'Reilly, 2001.

[10] Linux cross-reference project. Available on `http://lxr.linux.no/`.

[11] A. Mili, R. Mili, and R. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5:349–414, 1998.

[12] S. Overhage and P. Thomas. WS-Specification: Specifying web services using UDDI improvements. In *Web, Web-Services, and Database Systems*, volume 2593 of *LNCS*. Springer, 2003.

[13] Y. Padioleau and O. Ridoux. A logic file system. In *USENIX Annual Technical Conference*, 2003.

[14] Y. Padioleau and O. Ridoux. A parts-of-file file system. In *USENIX Annual Technical Conference*, 2005.

[15] R. Prieto-Díaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, 1987.

[16] M. Rittri. Using types as search keys in function libraries. In *4th Int. Conf. on Functional Programming Languages and Computer Architecture*. ACM Press, 1989.

[17] G. Ruhe. Intelligent support for selection of COTS products. In *Web, Web-Services, and Database Systems*, volume 2593 of *LNCS*. Springer, 2003.

[18] C. Runciman and I. Toyn. Retrieving re-usable software components by polymorphic type. In *4th Int. Conf. on Functional Programming Languages and Computer Architecture*. ACM Press, 1989.

[19] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, 1999.

[20] C. J. van Rijsbergen. A new theoretical framework for information retrieval. In *Int. Conf. on Research and Development in Information Retrieval*, 1986.