

# Use of the Linking File System to Aid the Study of Software Evolution

Sasha Ames

*sasha@cs.ucsc.edu*

CMPS290g Topics in Software Engineering: Evolution  
University of California, Santa Cruz

## Abstract

*This paper presents the use of the Linking File System as infrastructure for software evolution research. The addition of richer metadata in links and attributes makes LiFS an alternative to conventional approaches to storage for that purpose. We pursue our approach through the instrumentation of Kenyon and a case study that involves the correlation of Bugzilla data and CVS histories. Further investigation into usability and performance evaluation will need to follow.*

## 1. Introduction

A common requirement across many areas of research is that of sound infrastructure for data storage. The infrastructure should meet the needs of its users through satisfying certain criteria that include efficiency, functionality and usability. Moreover, users demand that simply storing ones data is no longer sufficient. They require adding metadata to describe the data and the ability to express semantic relationships about it all. We have found that traditional file systems themselves lack the framework to support those requirements, and generally solutions have been though ad hoc techniques [1].

We have certainly encountered needs along these lines in the study of software evolution. Many projects have been undertaken that involve the storage of historical data [3, 17, 8, 19, 11]. All the projects use conventional techniques for data and metadata storage, such as traditional file systems and relational databases. They use these systems for their infrastructure because nothing better has yet been made available to them. Furthermore, Gasser *et al.* have suggested a need to standardize metadata, tools, and provide a centralized data repository with federated access for software research [10]. Sound infrastructure will be needed to meet those demands.

In this paper, I present the Linking File System or LiFS as an alternative infrastructure for use in implementing tools for software evolution studies. LiFS provides efficient infrastructure that is easy to use. It adds functionality with links and attributes to conventional data storage. Thus, metadata may be placed directly with the data it describes for ease in access, as opposed to being placed separately in a database. The consequence of the latter option is that semantics might be lost if we cannot maintain the correlation between these conventional, separate systems over time. Additionally, LiFS can represent graph structures, which may be very helpful to software studies that use graph analysis, such as those mentioned in [14, 2, 12].

To demonstrate one aspect in which LiFS may aid the study of software evolution I present a case study application of LiFS. This study should seem familiar within the field of software evolution. It involves the correlation of bug records from a Bugzilla [4] and software revision history data in a CVS repository [6] for a small software project. The goal of this case study is to have a linking file system instance created for presentation of this data to a user who may want to see some specifics of the project with respect to the available data. This is similar to a goal of Hipikat, where the authors mention the desire to assist a newcomer to a project get up to speed through learning parts of its history [5].

This remainder of this paper is organized as follows. Section 2 gives an overview of the major features of the linking file system. Section 3 discusses the methodology and implementation for our proof of concept, namely the instrumentation of Kenyon. Section 4 presents the case study, composed of implementation details and specific example output. Section 5 mentions how we may evaluate the approach taken here and Section 6 suggests some threats to validity. Section 7 and 8 lists the related and future work, and section 9 concludes.

## 2. Overview of LiFS

In this section I give an overview of the major features of LiFS. We describe LiFS in more detail here [1], but I wish to convey the features that best distinguish it from conventional file systems.

LiFS contains a new file system feature, called a *relational link*. These relational links express relationships between any two files. Thus, in LiFS a regular file may point or refer to another. Relational links are different than symbolic links which allow a user to create additional path names for files or directories, thus becoming “linked” to additional places other than the original path. Moreover, relational links may contain metadata in the form of associated attributes.

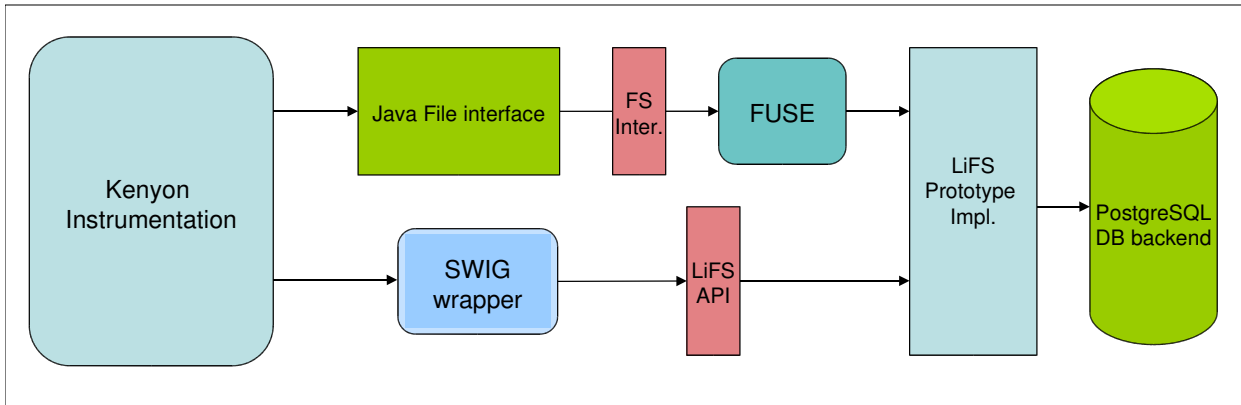
These attributes take the form of key/value pairs where on a given link, each key must be unique. LiFS also allows attributes placed on files directly, as is done using the extended attribute API in the Linux kernel 2.6.x [13].

The existence of links between files leads us to a loss of distinction between files and directories. Hence, having files refer to other files allows files to “contain” others, a property that was previously reserved for directories. Thus, directories become, in essence, simply zero byte files that exist for organizational purposes and backwards compatibility with conventional file system structures. Moreover, this implies that a path to any given file shall occur through a series of files that may also contain data and can be opened.

LiFS additionally allows multiple links between files. This is so a single or multiple users may wish to have different sets of metadata to show separate relationships between the two files. I will suggest an example use of this feature in section 4.2.4. The caveat to this feature is that links must be uniquely identifiable through some combination of their attributes. A simple identification scheme, but one not required by the file system, is to have a unique name attribute on each link.

The proposed implementation of LiFS is to have all metadata for files, links and attributes reside in magnetic RAM. MRAM is a new form of high speed, non-volatile RAM. It should greatly increase metadata access performance over conventional reads and writes to disk for metadata.

An example related to this research where LiFS might be used is for a the working directory of a software project. Binaries may be linked to object files, which in turn may be linked to the source files that generated them, or vice-versa. In addition, we can compile source code for different hardware architectures by following the appropriate link to the compiler configured for the target architecture, where a link exists to a compiler for each architecture. The resulting object



**Figure 1. The architecture of the LiFS prototype and how Kenyon may access it.**

code could be linked to the source with attributes indicating which architecture it should run on.

A major advantage to using LiFS for data organization is that related data from various sources may be linked together in the file system for the user’s appreciation, rather than utilizing some other ad hoc technique. My case study in section 4.1 shall present how we can relate the data extracted from a Bugzilla database to data concerning a project extracted from a CVS repository using links and attributes.

### 3. Methodology and Implementation

To demonstrate the application of LiFS for use in software evolution studies, I have decided to combine LiFS with Kenyon. Kenyon provides an excellent framework to work with, in that it has infrastructure already in place to extract data from SCM repositories. The data is then represented in a common internal format. Additionally, Kenyon uses `ConfigGraph` objects for representation of a graph structure [3].

The version of LiFS I used is not the one discussed above (section 2) that will utilize MRAM, but a prototype to serve as a proof of concept for LiFS. This prototype runs with a PostgreSQL database [18] as the backing store for LiFS’ metadata. The interface for standard file system operations comes through FUSE (file systems in user space) [16]. A separate channel to the file system

implementation. provides additional API to LiFS calls that involve access and manipulation of relational links. This API I have made available for use in programs written in Java via SWIG wrapper code [15]. This allows me to properly instrument Kenyon for use of LiFS.

We have a reasonably straightforward “instrumentation” of Kenyon. So, as we construct the graph, which Kenyon normally will store in a database via the Hibernate interface, instead we store the graph data in LiFS. Basically, methods associated with the `ConfigGraph` class and other related classes have additional calls added to them that interact with LiFS. The `addNode()` method creates a new file. The `addEdge()` method creates a link between two files already created by `addNode` calls. Within `Node` and `Edge` classes, the `addAttribute()` or `setAttribute()` methods have been instrumented to add or change the attributes on files or links. Additionally, we write the attribute data to the file (in text). This is necessary when utilizing the present LiFS implementation because there are issues in retrieving the attribute data set on files in the prototype.

We have two tools to retrieve this data for user inspection or analysis. We can explore LiFS in the command line using standard shell operations to the file system and a new program called `ls2`. `ls2` is a spin-off of `ls` that displays links with attributes for a given source file. Also, for graph-

ical browsing of the file system, one may use *LiFSBrowse*. *LiFSBrowse* shows the links interconnecting the files in LiFS with either a tree or graph structure display. It shows the attributes on a selected link and the content of the target file of that link. Additionally, it allows for some interaction with the file system, such as creation, modification of new links, and a query for links based on matching attributes. *LiFSBrowse* is still under development and should also improve greatly with the future LiFS implementation.

## 4. Case Study: File System development

I have selected the present ongoing development of the next version of LiFS for my case study. I see a number of key advantages from selecting this project over some other arbitrary open-source project. First, I am familiar with the project's history myself. Therefore, I should be able to easily verify if the results are sound and fit with what I should expect to see. Second, the scale of the project is good for presenting examples. The project has three subcomponents with roughly 10-30 files each and the time frame is over 6 weeks. This makes the extracted data very manageable from an evaluators perspective. Also, there are a small number of bug IDs in the Bugzilla:  $\sim 20$ . Finally, to ease the coupling between Bugzilla bugs and CVS transactions, I have instructed the project's developers to indicate to which bug ID a CVS commit relates in the log message. Thus, the coupling information becomes trivial to extract..

### 4.1. Fact Extractor Implementation

I implement the process of storing the CVS and Bugzilla data I wish to examine using a Kenyon fact extractor. A fact extractor is a subclass of the `FactExtractor` class that is part of the Kenyon code base. This fact extractor takes advantage of Kenyon's SCM

log parsing functionality. Kenyon provides all revisions from a single commit operation within a `Transaction` object. For each transaction found, my fact extractor, called the `CVSLogExtractor`, creates a new node. We save the metadata of the author name and log message in a link to the transaction node from a root node and on the node itself. Additionally, we parse the recorded Bugzilla bug ID from the log message so it may be used to link associated bug data to the transaction.

After that, we create a node for each file revision in the current transaction. We link the transaction nodes to the revision nodes and create attributes for the file name, file's revision number, and the time stamp. If a node already exists for the file, we link the current transaction to that node, as opposed to creating a new node. Node file names contain both the actual file name and a hash code for the parent file path, generated by the Java `hashCode()` method. This was necessary due to a problem with the prototype handling path names, resulting in storing all files in a flat space. Given that various directories within a project may contain different files with same name, for example, Makefiles, we wish to be able to differentiate between each under one directory, made possible by the unique hash code for each path..

Then, we provide additional information from the revision files. We apply this process only to `.c` and `.h` files; all others are ignored. We count the lines of code for each file. This is a simple count which includes blank lines and comments. Given that a file may persist through multiple transactions and each transaction will have links to the node for the file, we may see the number of lines of code vary from one link to the next. The other information we acquire from reading the file is links for `#include` directives within source files. We only process local `#includes` for `.c` and header files within the same directory. For each `#include` we create a new link from the source file containing the `#include`

to the target included file. This builds a subgraph of `#include` statements for a group of related files. As we repeat this process for each transaction over time, we may see new links for new header file inclusions. This should allow us to see the evolution of `#includes` within the software project.

The second part of the fact extractor implementation is a component that creates files and links for the Bugzilla data. It should be noted that Bugzilla bugs do not necessarily refer strictly to “bugs” to be fixed, but also new feature or enhancement requests. We create a file (Node) for each Bugzilla bug record. We link a root node for the bug records, named “AllBugs” to each bug node and place the bug record metadata on both the link and the node itself. This metadata includes the bug ID, the owner, the bug priority, severity, current status, the component, and a summary of the issue. Additionally, we create links between the bug nodes when we encounter relationships given between bug records. Thus, when bug 2 depends on the completion of bug 1, we will create a link from bug 2 to 1 tagged with an attribute to specify that relationship.

We perform this process prior to the extraction of the SCM data and creation of the corresponding graph components. Thus, when we go through the transactions and extract the bug IDs, we can link the bug record nodes to the appropriate SCM transactions that address that particular bug record. With that, the creation of a linking file system to represent the history of the project is complete.

## 4.2. Examples of Generated LiFS Data

I have generated a linking file system from the case study data with which I shall show examples of how we may use links to learn a few things about our subject matter. This data is generated with Kenyon configured to extract from the CVS repository with a one week period. This has resulted in only three transactions extracted,

although there are more transactions that have occurred during the specified time frame. I did attempt to run Kenyon with a shorter frequency, but the resulting data caused problems with the LiFS prototype that were difficult to diagnose. Therefore, I chose to make my examples with the weekly extraction data, as it did produce stable results. Nonetheless, these results I believe do give a picture of some of the history of this project.

There are two views given for the examples from the project. First are screen-shots from executions of `ls2` run on various files within the resulting file system. The images contain links, each with a source file, the target files for that source, and the attributes shown as comma delimited strings of key/value pairs. Also, I have provided screen-shots of related LiFSBrowse views. In those images, the left-hand column shows the linking file structure, i.e. a graph rendering of the files and links. Files are shown by blue circles, and links are shown with black arrows or arcs connecting the files. The label below or on each link gives the file name for the target file for that link. A link and its label are together highlighted using the color red when the user selects that link. The upper left quadrant of the application shows the attributes on a selected link, and the lower right quadrant shows the content of the target file of the selected link, if a “file\_type” attribute designates it as text.

### 4.2.1. Showing Bugs and Transactions

When examining a file system generated for a project like our case study, the first items that a user may consider are the root nodes or files. These files contain the links to files for individual transactions and bug records. Figure 2 shows the links from these root nodes with the metadata shown as comma delimited strings for each link. Figure 3 also shows the same links graphically. In both figures we can see that there are a large number of links emanating from `AllBugs.node`. Given that there is significant metadata on each

```

[sasha@mram1 flat0]$ ls2 ROOT.node
ROOT.node
-> /flat/transaction3.node
  - edgeName=trans_id:1,userId=500
-> /flat/transaction1.node
  - authors=sasha,edgeName=trans_id:1,file_type=text,userId=500
-> /flat/transaction2.node
  - edgeName=trans_id:1,userId=500
[sasha@mram1 flat0]$ ls2 AllBugs.node
AllBugs.node
-> /flat/BUG:12.node
  - Component=Kernel files,edgeName=BUG:12,file_type=text,Id=12,Owner=sasha@soe.ucsc.edu,Priority=P2,Severity=critical,Status=RESOLVED,userId=500
-> /flat/BUG:5.node
  - Component=Kernel files,edgeName=BUG:5,file_type=text,Id=5,Owner=sgoldreid@soe.ucsc.edu,Priority=P2,Severity=normal,Status=RESOLVED,Summary=more syscalls to kernel,userId=500
-> /flat/BUG:9.node
  - Component=File Storage,edgeName=BUG:9,file_type=text,Id=9,Owner=sasha@soe.ucsc.edu,Priority=P2,Severity=normal,Status=RESOLVED,Summary=first cut of storage module,userId=500
-> /flat/BUG:13.node
  - Component=File Storage,edgeName=BUG:13,file_type=text,Id=13,Owner=sasha@soe.ucsc.edu,Priority=P2,Severity=normal,Status=REOPENED,Summary=make changes,userId=500
-> /flat/BUG:7.node
  - Component=Fuse mod,edgeName=BUG:7,file_type=text,Id=7,Owner=sasha@soe.ucsc.edu,Priority=P2,Severity=normal,Status=RESOLVED,Summary=Fuse calls for getting linksets,userId=500
-> /flat/BUG:15.node
  - Component=File Storage,edgeName=BUG:15,file_type=text,Id=15,Owner=sasha@soe.ucsc.edu,Priority=P3,Severity=enhancement,Status=NEW,Summary=handle sparse files,userId=500
-> /flat/BUG:18.node
  - Component=System calls impl.,edgeName=BUG:18,file_type=text,Id=18,Owner=nikhil@soe.ucsc.edu,Priority=P2,Severity=normal,Status=NEW,Summary=implement some syscalls,userId=500
-> /flat/BUG:16.node
  - Component=System calls impl.,edgeName=BUG:16,file_type=text,Id=16,Owner=sasha@soe.ucsc.edu,Priority=P5,Severity=enhancement,Status=ASSIGNED,Summary=improve attribute set performance,userId=500
-> /flat/BUG:8.node
  - Component=Linking/,edgeName=BUG:8,file_type=text,Id=8,Owner=nikhil@soe.ucsc.edu,Priority=P2,Severity=normal,Status=NEW,Summary=get started committing your code,userId=500

```

Figure 2. ls2 view of bug and transaction root nodes and links. Explained in section 4.2.1.

link, the use of an attribute based query might be helpful to the user in order to narrow down the links displayed to bug records that she may consider relevant.

**4.2.2. Links from Bugs** Figure 4 shows the links that have the source file for bugs 2 and 5. Bug 2 has a single link to bug 1. Bug 5 has two different types of links as implied by that the attribute strings on each have different key names. The “relationship” key name is common across both links, but the link between bugs has the value “dependsOn”, while the link to the transaction node differs. While the other links described in section 4.2.1 contain metadata describing the node to which they point, these links do not. They exist more to show the relationships between the nodes of same or different types.

Figure 5 gives a graphical view of some links similar to those described above. In the bottom half of the left-hand side column we see the nodes

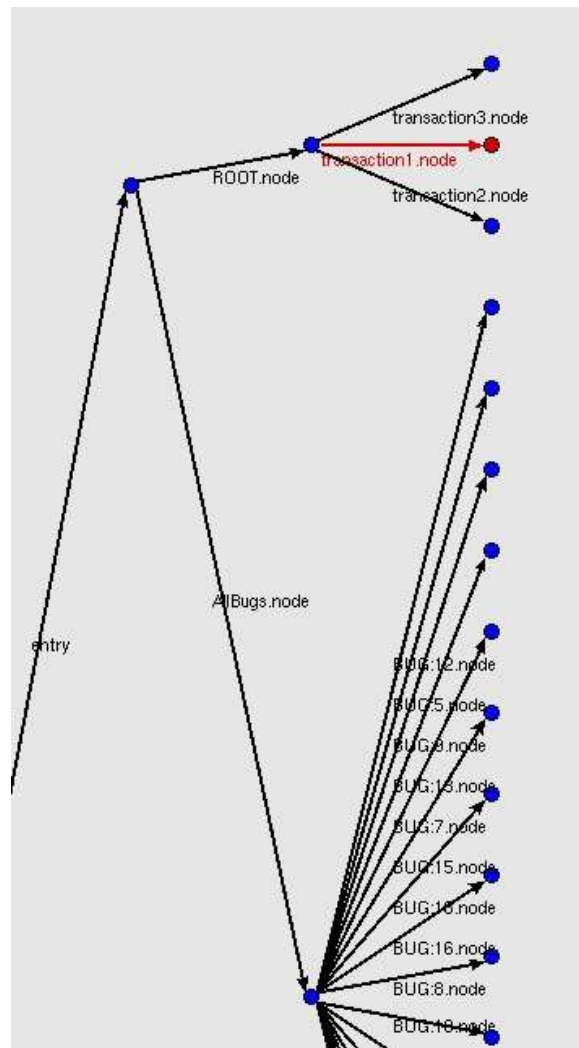


Figure 3. LiFSBrowse view of bug and transaction root nodes and links. Explained in section 4.2.1.

```

[sasha@mram1 flat0]$ ls2 BUG:5.node
BUG:5.node
-> /flat/BUG:2.node
  - edgeName=bugs5to2,file_type=text,relationship=dependsOn,userId=500
-> /flat/transaction2.node
  - BugId=5,edgeName=bugstrans5to2,file_type=text,relationship=revisions,Transaction=2,userId=500
[sasha@mram1 flat0]$ ls2 BUG2.node
ls2: BUG2.node: No such file or directory
[sasha@mram1 flat0]$ ls2 BUG:2.node
BUG:2.node
-> /flat/BUG:1.node
  - edgeName=bugs2to1,file_type=text,relationship=dependsOn,userId=500
[sasha@mram1 flat0]$

```

Figure 4. ls2 view of links from bug records to other bugs and transactions. Explained in section 4.2.2.

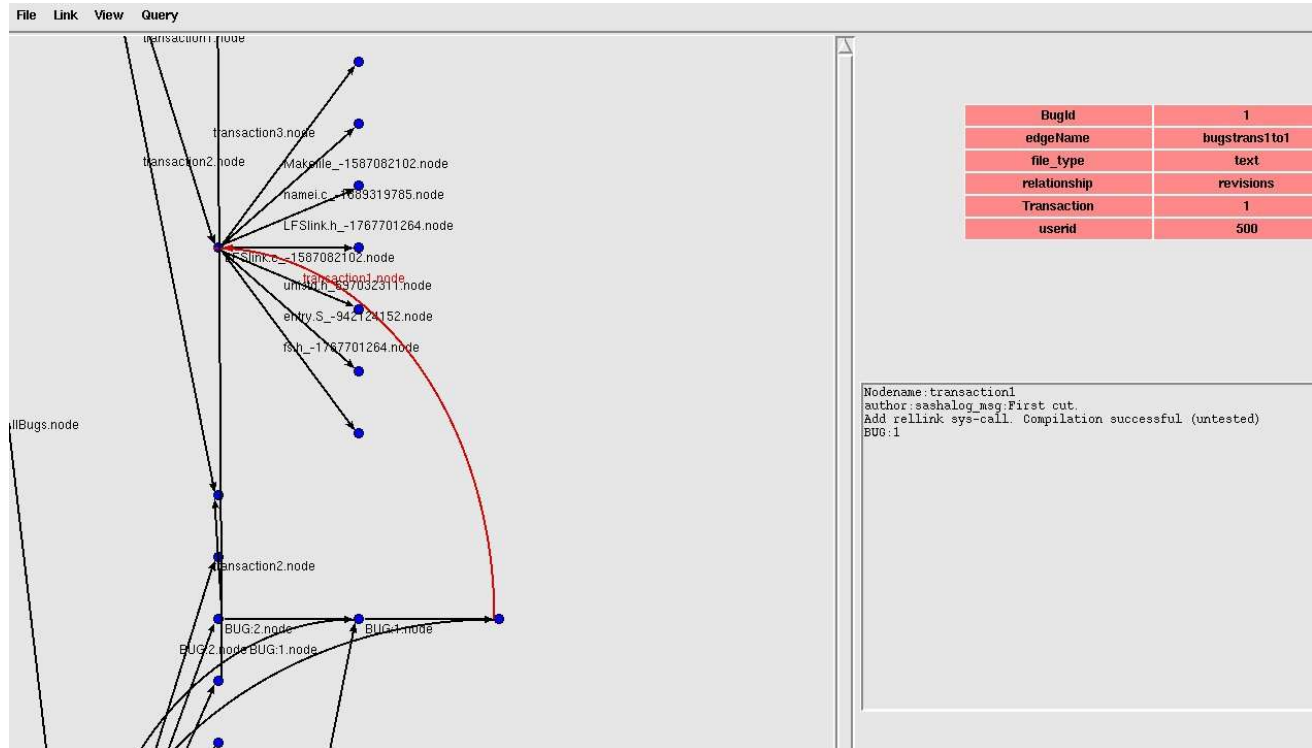


Figure 5. LiFSBrowse view of links from bug records to other bugs and transactions. Explained in section 4.2.2.

for bug records. The arc hi-lighted red goes from the file for bug 1 to file for transaction 1. We see in the upper left that the relationship is “revisions”<sup>1</sup> and the transaction is number 1. The lower right corner shows the metadata written to the file for that node. It contains the author’s name and log message. The transaction node also has a number of links from it, of which I shall describe next.

**4.2.3. Revisions** Figure 6 gives a list of the links from a transaction node file to the revision files. All the links have consistent metadata, i.e., similar sets of keys on each. On this transaction we may note that each file revision has the same timestamp, suggesting it took place with a single commit statement. However, we can see different

```

[sasha@mrani Flat0]$ ls2 transaction2.node
transaction2.node
-> /flat/namei.c_-1889319785.node
  - date=Thu Feb 10 01:16:25 PST 2005,edgeName=namei.c_-1889319785_2,filena
me=namei.c,lines_of_code=2377,revnum=1.3,trans_id=2,userid=500
-> /flat/file.c_-1889319785.node
  - date=Thu Feb 10 01:16:25 PST 2005,edgeName=file.c_-1889319785_2,filenam
e=file.c,lines_of_code=146,revnum=1.1,trans_id=2,userid=500
-> /flat/LFSlink.c_-1587082102.node
  - date=Thu Feb 10 01:16:25 PST 2005,edgeName=LFSlink.c_-1587082102_2,file
names=LFSlink.c,lines_of_code=231,revnum=1.3,trans_id=2,userid=500
-> /flat/unistd.h_597032311.node
  - date=Thu Feb 10 01:16:25 PST 2005,edgeName=unistd.h_597032311_2,filenam
e=unistd.h,lines_of_code=474,revnum=1.3,trans_id=2,userid=500
-> /flat/lfslink.c_-1889319785.node
  - date=Thu Feb 10 01:16:25 PST 2005,edgeName=lfslink.c_-1889319785_2,file
name=lfslink.c,lines_of_code=38,revnum=1.1,trans_id=2,userid=500
-> /flat/entry.S_-942124152.node
  - date=Thu Feb 10 01:16:25 PST 2005,edgeName=entry.S_-942124152_2,filenam
e=entry.S,revnum=1.2,trans_id=2,userid=500
-> /flat/fs.h_-1767701264.node
  - date=Thu Feb 10 01:16:25 PST 2005,edgeName=fs.h_-1767701264_2,filenames=
fs.h,lines_of_code=1653,revnum=1.3,trans_id=2,userid=500
-> /flat/lfslink.h_-1889319785.node
  - date=Thu Feb 10 01:16:25 PST 2005,edgeName=lfslink.h_-1889319785_2,file
name=lfslink.h,lines_of_code=4,revnum=1.1,trans_id=2,userid=500
-> /flat/Makefile_-1889319785.node
  - date=Thu Feb 10 01:16:25 PST 2005,edgeName=Makefile_-1889319785_2,filen
ame=Makefile,revnum=1.1,trans_id=2,userid=500
[sasha@mrani Flat0]$

```

Figure 6. Links to file revisions from a transaction node. Explained in section 4.2.3.

version numbers, so some of the files are likely to have been changed through other transactions

<sup>1</sup>“Transaction” may have been a better name for the value for this relationship attribute.

```

[sasha@mraml flat0]$ ls2 interface.c*
interface.c_-1893301018.node
-> /flat/interface.h_-1893301018.node
- directive=#include,edgeName=include:interface.c_-1893301018:interface.h_-1893301018_3,filename=interface.h,file_type=text,transaction=3,userid=500
[sasha@mraml flat0]$ ls2 interface.h*
interface.h_-1893301018.node
-> /flat/disk_io.h_-1893301018.node
- directive=#include,edgeName=include:interface.h_-1893301018:disk_io.h_-1893301018_3,filename=disk_io.h,file_type=text,transaction=3,userid=500
-> /flat/disk_alloc.h_-1893301018.node
- directive=#include,edgeName=include:interface.h_-1893301018:disk_alloc.h_-1893301018_3,filename=disk_alloc.h,file_type=text,transaction=3,userid=500
-> /flat/stubs.h_-1893301018.node
- directive=#include,edgeName=include:interface.h_-1893301018:stubs.h_-1893301018_3,filename=stubs.h,file_type=text,transaction=3,userid=500

```

**Figure 7. Sample of links within the #include graph. Explained in section 4.2.4.**

with differing frequencies. After each file name, we see the hash code for the parent path name. Some have matching codes, indicating that they are from the same source directory, while others do not.

Although only a single transaction is represented here, we may learn additional information from looking at multiple transactions that reference some of the same files. The links pointing to the same files from different transactions should have different timestamps, revision numbers, and number of lines of code. This should infer the frequency of changes and if an addition or subtraction to the file content has occurred. Additional metrics could be placed on the links for revisions to show changes over time beyond what I have done for this case study.

**4.2.4. Include Graph Represented** Figure 7 gives some links that constitute part of the #include subgraph for a group of files. The `interface.c` file has a single link to `interface.h`, and subsequently, `interface.h` has a few links to some other header files. We may deduce through a path of links what code from which header files ends up included within a `.c` source file. Also, this data includes the transaction number on each link. In this case the transaction number is always the same, as only a single transaction was used to generate this include graph. However, it is possible that we could have used multiple transac-

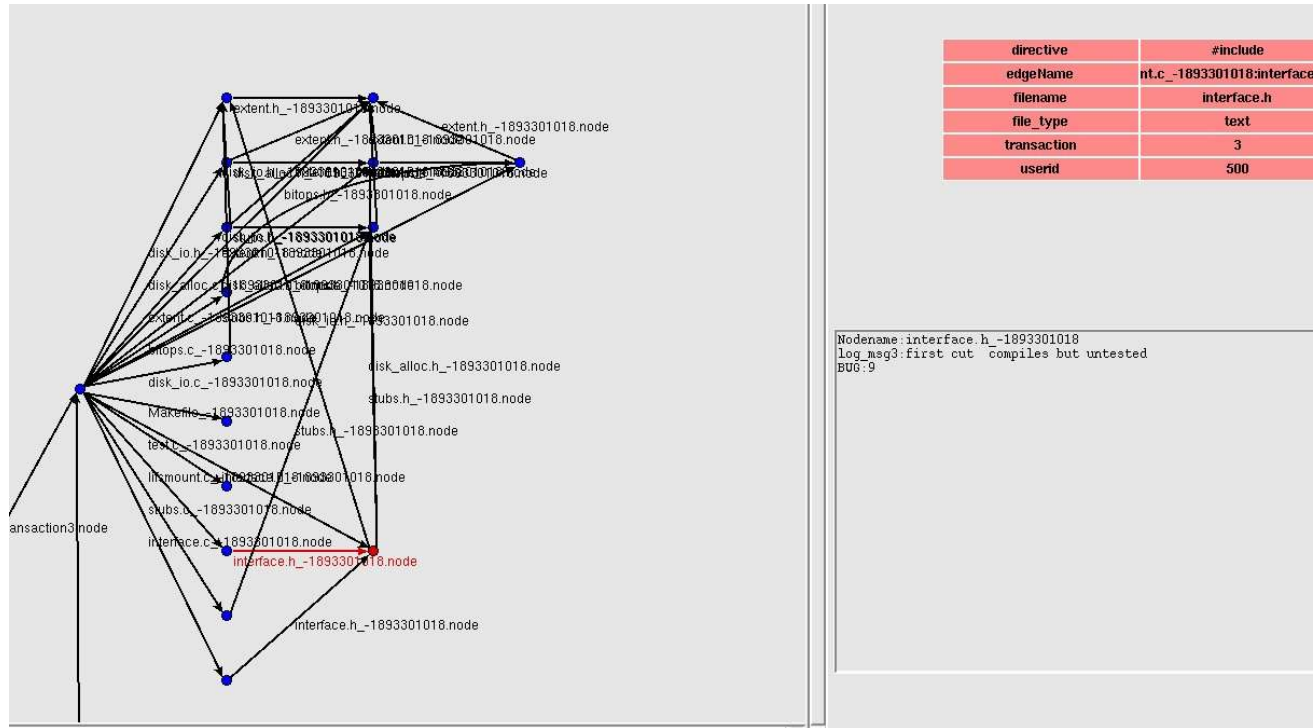
tions to generate the graph. In that case we may have a situation in which we encounter multiple links between files: one per transaction where the #include directive is present between files. Increases in the number of links over time may indicate growth in functionality or increased coupling between modules. Decreases may indicate refactoring performed between some transactions.

Figure 8 shows the LiFSBrowse view of the include graph. This may aid users trace what headers are included in which source files and see the overall complexity of header file inclusion within a project. We may identify which header files define the lowest level operations in a web of dependencies from viewing this graph. Querying by the transaction number attribute may show how the dependencies have evolved as well.

**4.2.5. Whole File System** Figure 9 lets us see an overview of the structure of the entire generated file system. LiFSBrowse runs here in its maximum “zoom out” mode. The main advantage of this view is to give a sense of the number of bugs linked to transactions, The long links from the lower to up half connect the bug records to various transactions. The image shows the link connecting “bug 9” with “transaction 3” highlighted. The log message associated with the transaction reveals that the author should probably try to write slightly more descriptive messages, as one may not be aware of what the first cut accomplishes.

## 5. Evaluation

I wish to evaluate how LiFS stands as sound infrastructure to aid in software evolution research through two means. This section shall propose how to go about such evaluation. First is usability. The examples presented in the case study above (section 4.2) show one such use for LiFS. While I may claim that the examples speak for themselves with respect to usability, they are severely limited by the tools we have to explore



**Figure 8. LiFSBrowse view of #include graph. Explained in section 4.2.4.**

the file system. For instance, LiFSBrowse could benefit enormously from an improved graph layout engine. Presenting examples like to a group of users for evaluation at this point would thus not really reflect the use of LiFS itself, but the applications that run atop it.

Moreover, the ease in implementing tools that would use LiFS as a repository may figure into the value of LiFS to a researcher. I may wish to claim that it will be easier to work with the LiFS API than API to access a database to conduct S.E. research, as both can be used to represent the same data. I may add an additional claim to the value of LiFS through stating that LiFS facilitates this by first, placing the data in the file system so it is more easily accessible to a user, and second, combines the file data and metadata together for ease in accessibility. Of course, these claims are highly subjective, and evaluation may only favor one view over another. We may wish to experiment with human subjects and see to what extent they may favor LiFS over conventional ap-

proaches. Such studies do not seem like a particularly easy task.

The other aspect of LiFS to evaluate is performance. Good performance is a common goal of much research, with systems research in particular. We would want to look at the performance overhead of instrumenting Kenyon and extracting data from LiFS versus using a database repository. Such evaluations are better left until we have a proper version of LiFS to benchmark, as we are already aware that the prototype implementation of LiFS does not perform very well. My initial observations of the instrumentation runs shows little noticeable loss in performance for Kenyon, but complex query operations were not involved.

## 6. Threats to Validity

This exploration of the use of the file system is still very preliminary. At this point my study lacks results of evaluation as mentioned above

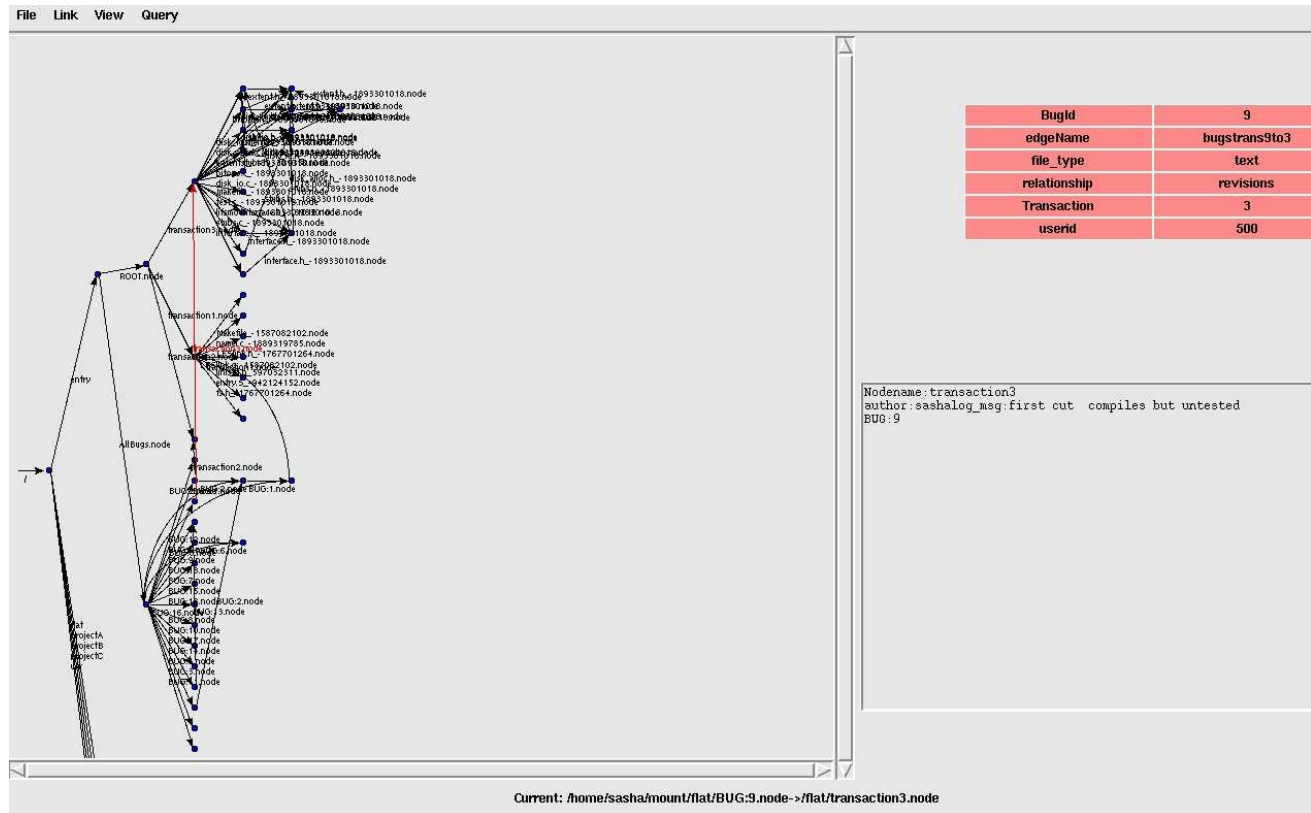


Figure 9. LiFSBrowse view of the entire generated file system. Explained in section 4.2.5.

in section 5. My use of the prototype has constrained the scope of this research. An example of this is that I was limited to short runs of Kenyon for extraction of my CVS repository. Moreover, I only considered a single project for a case study and used a single fact extractor.

## 7. Related Work

Hipikat is a system with similar goals as those for my case study [5]. Its authors had in mind assisting new developers get up to speed with a project through giving an understanding of its artifacts. The system allows users to add certain artifacts to their system such as file revisions, bug records or documents. These are similar to the files or nodes added to LiFS in the case study. Users may then add links between the artifacts using Hipikat. Additionally, Hipikat contains a

UI to allow users to browse the system, akin to LiFSBrowse.

Whereas LiFSBrowse allows users to visualize graph structures in LiFS, there has been some work done in visualizing software history. Gall, *et al.* looked at using 3-D pictures of module decompositions and repository changes over time [9]. LiFSBrowse allows for visualization of the same data, but without the use of the third dimension. Eick, *et al.* show the use of a graphical tool to show relationships between files based on modification requests [7]. Likewise, we may present a similar picture when browsing the LiFS implementation case study.

A series of work relates to populating SCM and bug tracking data into a common repository. Fischer, *et al.* consider the use of CVS and Bugzilla in their work to populate their RHDB [8]. SoftChange similarly has a repos-

itory for CVS and Bugzilla data [11]. It additionally considers ChangeLog data and discusses a process for recovering MRs, much like the `Transaction` objects in Kenyon that we store using LiFS. Zimmerman also looks at extracting CVS data, mirroring it in a database, and restoring the individual transaction groupings of revisions [19]. Finally, BEAGLE contains a database to store mined SCM data [17]. While the LiFS prototype implementation presently also uses a database to store its file system representation, we hope that it will eventually serve as a different approach to storing this type of data, that being a high-performance in-memory file system.

## 8. Future Work

While in many ways the work presented here so far is that of a proof of concept, future work on this should move our understanding beyond that stage. First and foremost, the next generation LiFS implementation should hopefully address all problems that we have found using the prototype. As shown in section 4.2 I had to do a somewhat limited data extraction with Kenyon due to problems with the prototype that we believe are rooted in how it was integrated with PostgreSQL. Robust scalability is a major goal for the next version, and we hope it should correct the problem. Additionally, other bugs that affected this project, including the inability to retrieve the attributes on files, should be corrected as well.

Another potential feature considered for this project was to use LiFS directly as the target file system for Kenyon’s SCM extraction, and this too was hampered by flaws in the prototype. By default, we configure Kenyon to extract the data to `/tmp`, but if a user wishes to preserve and perhaps manipulate the data, which could include the application of LiFS features directly (links and attributes), it would be possible to extract the files directly to a LiFS directory instead of to `/tmp`. This is unfortunately not pos-

sible with the LiFS prototype because pathnames within an instance of LiFS must have directory names flagged with an `@` character. My instrumentation of Kenyon contained this “fudge” to make it work. The output from the SCM extraction process usually contains somewhat complex directory structures. Further instrumentation of Kenyon would be required to make it work. Thus, I decided to postpone implementing this feature, as the next version of LiFS should solve this problem. Then, I shall be able to write a fact extractor that could work with the data written directly to LiFS.

LiFS could potentially facilitate the administration of “Time-travel builds”. Such allows an investigator to build different versions of a product over time through extracting those versions from an SCM and perform the builds. While this is certainly possible using conventional file systems, LiFS should facilitate this through providing the infrastructure for metadata regarding each individual build, providing the linking of the built version with the source files, and potentially saving space through building a directory structure using links for each build version. Each structure can point to common versions file that are shared cross build releases. In addition, where we may find that different versions of a compiler or build tools have been used for various versions, they can be linked to the version under consideration as well.

Finally, LiFS allows for the construction of compound documents. LiFS may represent a compound document through a file containing links to other files with which to concatenate, or perhaps each file containing a single piece of the larger document may link to the next piece, and so on, to form the entire document. As much of the study of software evolution comes from seeing how “deltas” in SCMs change over time, it could be possible to have the different versions of files under consideration be represented through compound documents as sequences of the deltas, held together using links.

## 9. Conclusion

We have shown that alternatives to conventional approaches to metadata storage can be adopted for use in software evolution research. In our case, the task has been simplified by chance. The two existing systems we consider happen to share similarity in a common graph structure that allows them to be integrated. We see this in our instrumentation of Kenyon with calls to the LiFS API. We have shown that LiFS can be used for showing relationships between data. This application is most valuable when situations arise where we encounter related data from disparate sources. The case of Bugzilla and CVS data effectively demonstrates this usage.

Work on LiFS is still ongoing with the prototype as a start. It served the purpose of identifying possible applications of LiFS, but falls short for purposes of evaluation where we already know that performance will be sub-par. Additionally, further investigation into the true usability of LiFS versus conventional techniques from an application programmer's point of view will be necessary to truly give light to the value of our infrastructure.

## Acknowledgments

I would like to say thank you to Sung Kim and Jennifer Bevan for their assistance in using Kenyon. Additional thanks to the MRAM project group at the Storage Systems Research Center of UCSC for their many ideas that may have influenced my work on this investigation. Research for LiFS was funded in part by National Science Foundation grant 0306650.

## References

[1] A. Ames, N. Bobb, S. A. Brandt, A. Hiatt, C. Maltzahn, E. L. Miller, A. Neeman, and D. Tuteja. Richer file system metadata using links and attributes. In *Proceedings of the*

*22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, Monterey, CA, Apr. 2005.

- [2] J. Bevan and J. E. James Whitehead. Identification of software instabilities. In *2003 Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, Canada, November 2003.
- [3] J. Bevan, S. Kim, and L. Zou. Kenyon: A common software stratigraphy system. <http://www.soe.ucsc.edu/labs/grase/kenyon/>, 2004.
- [4] Home :: Bugzilla :: [bugzilla.org](http://www.bugzilla.org/). <http://www.bugzilla.org/>, 2005.
- [5] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *2003 Int'l Conference on Software Engineering (ICSE 2003)*, pages 408–418, Portland, OR, 2003.
- [6] Welcome to cvs home! <https://www.cvshome.org/>, 2005.
- [7] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *IEEE Trans. Software Engineering*, 28(4):396–412, April 2002.
- [8] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *2003 Int'l Conference on Software Maintenance (ICSM'03)*, pages 23–32, September 2003.
- [9] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *International Conference on Software Maintenance (ICSM)*, pages 99–108, Oxford, England, August/September 1999.
- [10] L. Gasser, G. Ripoché, and R. J. Sandusky. Research infrastructure for empirical science of f/oss. In *Int'l Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, May 2004.
- [11] D. M. German. Mining cvs repositories, the softchange experience. In *Int'l Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, May 2004.
- [12] M. J. Harrold and B. Malloy. A unified interprocedural program representation for a maintenance environment. *IEEE Transactions on Software Engineering*, 19(6):584–593, June 1993.

- [13] The linux kernel archive.  
<http://www.kernel.org/>, 2005.
- [14] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.
- [15] Simplified wrapper and interface generator.  
<http://www.swig.org/>, 2005.
- [16] M. Szeredi. File System in User Space README.  
<http://www.stillhq.com/extracted/fuse/README>, 2003.
- [17] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *2002 Intl. Workshop on Program Comprehension (IWPC 2002)*, Paris, June 2002.
- [18] J. C. Worsley and J. D. Drake. *Practical PostgreSQL*. O’Reilly, 1st edition, 2002.
- [19] T. Zimmermann and P. Weigerber. Preprocessing cvs data for fine-grained analysis. In *Int’l Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, May 2004.