Textual Allusions to Artifacts in Software-related Repositories

Gina Venolia
Microsoft Research
One Microsoft Way
Redmond, WA 98052 USA
http://research.microsoft.com/~ginav
gina.venolia@microsoft.com

ABSTRACT
Much of what is written about a software project is soon forgotten. Software repositories are full of valuable information about the project: Bug descriptions, check-in messages, email and newsgroup archives, specifications, design documents, product documentation, and product support logs contain a wealth of information that can potentially help software developers resolve crucial questions about the history, rationale, and future plans for source code. For a variety of reasons, developers rarely turn to these resources when trying to answer these questions. We are building a full-text search that encompasses multiple repositories. To effectively implement full-text search in the absence of hyperlinks we propose detecting *textual allusions* to software artifacts in natural-language prose. Allusions are shown to contribute a significant portion of the relationships represented in the graph.

Categories and Subject Descriptors
H.3.1 [**Information storage and retrieval**]: Content Analysis and Indexing; H.3.3 [**Information storage and retrieval**]: Information search and retrieval—*Retrieval models*; H.5.3 [**Information interfaces and presentation**]: Group and Organization Interfaces—*Computer-supported cooperative work*; K.6.3 [**Management of computing and information systems**]: Software Management—*Software development, Software maintenance*.

General Terms
Documentation, Experimentation, Human Factors.

Keywords
Software development, project memory, software artifacts, search.

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

1. INTRODUCTION
In a series of surveys and interviews at Microsoft, my team learned that the most serious problem that software developers face is "understanding the rationale behind a piece of code" [4]. It's likely that this is a universal phenomenon, not limited to Microsoft. There are vast information repositories—bug descriptions, check-in messages, email and newsgroup archives, specifications, design documents, product documentation, product support logs, etc.—that have the potential to answer questions about rationale, but we found that developers rarely access them. Instead they examine the source code text and probe it in the debugger, and if those fail, they typically initiate a face-to-face conversation with the person they think will know the answer. This investigation process, while often successful, costs precious time and causes interruptions.

There are many good reasons for a developer to neglect the electronic repositories when trying to understand code. The developer does not know *a priori* whether a topic of interest is addressed in any repository. Fast full-text search is not implemented for all the repositories. Each repository has its own search and browse tools, and there's little consistency among them. It may be difficult to formulate a full-text query for the topic of interest, or to browse meaningfully for artifacts related to it. It may be difficult to assess whether an artifact (found by searching or browsing) is the last word on the topic or is hopelessly out-of-date. Given these barriers it's easy to understand why developers choose to neglect the electronic repositories.

To address these deficiencies developers need a good full-text search tool that spans the relevant repositories. Modern full-text search systems rely in part on link-analysis scoring algorithms, such as PageRank [5] and HITS [3], which estimate each artifact's importance based on analysis of the hyperlinks among the artifacts. Unfortunately hyperlinks are rare among software artifacts. This paper presents an approach to simulating hyperlinks by detecting *textual allusions* to software artifacts in the natural-language prose that is already present in many software artifacts.

1.1 Related work
The Hipikat system [1] provides artifact-based search for code-related artifacts. It combines structural relationships with relationships found by a measure of textual similarity.

Team Tracks [2], a recommender system for methods in source code, relies on implicit relationships discovered by aggregating

developers' navigation patterns in code to compute its recommendations.

2. REPRESENTING THE GRAPH

There are many types of software-related artifacts: source code files, classes, methods, bugs, check-ins, emails, specs, etc. While there are some common attributes across artifacts (e.g. a name, a brief description, and a date that the artifact first came into existence) each type may also have properties particular to it. Likewise there are many types of relationships: member-of, implements, mentions, addressed-to, etc. Together these requirements suggest that an appropriate representation may be a directed multi-graph where the nodes (representing the artifacts) and arcs (relationships) are typed.

Each node and arc in the index has an identifier that may be derived directly from its type and its identifying properties. In our present implementation each node has an *identifier*, which is a string composed of its type name and its identifying properties. For example the bug #12345 in the *AppBugs* database might be named "bug:appbugs:12345". The utility of this identifier will be covered in greater detail later in this section.

In the current implementation the nodes and arcs share a common abstract base class, `Entry`, properties for a unique identifier for the entry and the dates that the entry was created and deleted, which may be unspecified. Artifacts, or nodes in the graph, are represented by the abstract `Item` type, which derives from `Entry`. It extends `Entry` with properties for the human-readable name for the item, the string on which full-text search operates, and a URL or command for opening a viewer on the actual artifact (all are optional). Relationships, or arcs in the graph, are represented by the abstract `Link` type, which derives from `Entry`. It extends `Entry` with properties for the identifiers of the endpoint items, and an optional estimate of the confidence in the relationship.

The graph is represented in a persistent store called the *index*. The index is expected to be very large, so it is expected to be deployed as a shared resource. For these reasons the index is stored as a database using Microsoft SQL Server 2005. Stored procedures and web service APIs provide mechanisms for submitting, fetching, and deleting entries and for executing queries and retrieving results. Full-text search indexing is enabled for the `Item` name and search text columns.

When an entry is submitted to the index, the index first checks whether it already contains one with that identifier. If not then one is created with the specified property values; otherwise any newly-specified optional property values override the old ones. (The rationale for this behavior will be explained in section 3.4.)

This infrastructure provides a generic base on which a rich index of domain-specific information can be built.

3. PROVISIONING THE INDEX

An index is provisioned with artifacts and relationships from a collection of information repositories, e.g. bug databases, source code control system databases, email archives, etc. There are three distinct categories of provisioning—source schema, file structure, and textual allusions—discussed in the next three subsections.

3.1 Crawling the Source Schema

Each of the data sources has a unique programmatic interface, requiring custom software we call a *crawler*. The crawler uses the repository's interface to query for new information, creates instances of types derived from Item and Link to represent the new information, and then submits the instances to the index. The crawlers are run on a periodic basis, though in principle a crawler could be run when notified of new data by the data source.

For example consider the source-schema entries created by a crawler on a particular source code control system database. With this particular source code control system, check-ins are numbered sequentially. The crawler keeps a record of the last crawled check-in. When it runs, it queries the repository for the subsequent check-in numbers, and then iterates through each one, requesting detailed information about it. An instance of `CheckInItem` (i.e. the type derived from Item that represents a check-in) is created, along with a `DomainAccountItem` to represent the author, and an `AuthorLink` that connects the two items. Then, for each file revision in the check-in, the crawler created a `RevisionItem`, which is associated with the `CheckInItem` using a `ChangeLink`. A file revision is conceptually bound to a particular file path, so the crawler creates items to represent file itself and the directory hierarchy above it, associated with a chain of `ContainsLink` instances. The consecutive revisions are linked—a `RevisionItem` representing the previous revision is created and a `NextLink` relating it to the present `RevisionItem`.

Our current implementation has crawlers for the source code control system, the bug database, and email. In the future we expect to implement crawlers for Active Directory (the company-wide database that stores organization chart, email discussion list membership, and security group membership), a file system crawler, a website crawler, an RSS/Atom crawler for gathering weblogs, and perhaps others.

3.2 Cracking Structured Files

Structured files are an important source of artifacts and relationships in the index. Files occur many places in the repositories, e.g. as an attachment to an email or bug, or stored in a source code control system, file server, or web server. When a file is encountered it its type is used to look up a *cracker*, a piece of code that reads the file and produces items and links to represent its contents. For example a source code file contains useful structure, as does an XML file that controls a build process; on the other hand a Microsoft Word document has structure, but none that relates specifically to software-related artifacts.

The cracker "cracks open" the file and creates entries to represent its structure. The C# cracker runs a compiler front-end and walks the resulting abstract syntax tree to produce `ClassTypeItem` instances, `ImplementsLink` relationships to other `ClassTypeItem` instances, `FieldMemberItem` and `MethodMemberItem` instances and `ContainsLink` instances to associate them with the `ClassTypeItem`, etc. A build-file cracker would associate the items representing various source files with the item representing the binary output file.

Our current implementation has crackers for C/C++ files, which create shallow structure, C# files, which creates deeper structure, and any files for which an IFilter can be found (IFilter is public

Table 1: The number of items of each type, and their average number of incident arcs.

Type	Count	Avg. Degree
SCCS* file revision	2,688,714	2
SCCS file	878,736	3
SCCS check-in	379,913	8
SCCS folder	243,756	5
Identifier	190,177	4
Number	148,498	4
Bug	93,554	6
Bug revision	49,731	12
Local file path	17,436	3
Email message	12,203	47
HTTP URL	11,377	6
Email conversation	8,292	2
Domain account	8,067	70
Server file path	3,823	2
Email address	266	43
SCCS database	17	22,493
Bug database	4	23,388

* Source code control system

interface for components that convert files to plain-text streams; there are IFilters for dozens of file formats, including Microsoft Word, Excel, and PowerPoint, and Adobe PDF), which creates no structure but extracts a plain-text version of the file's contents, making it searchable and allowing it to be scanned for textual allusions. In the future we expect to implement a deeper cracker for C/C++, a cracker for Visual Studio project files, crackers for binary files, and perhaps others.

3.3 Scanning for Textual Allusions

Any text that is presumed to be natural-language prose that a crawler or cracker encounters is submitted to a battery of *scanners*, which scan the text for textual allusions to software-related artifacts and create entries to represent them. Each check-in crawled by the source code control system crawler has a check-in message which typically contains prose, as do comments in C++ source code, emails, word processing documents, and web pages. Each item type may contribute a scanner to the battery. We currently implement several scanners:

`EmailAddressItem`: Email addresses, e.g. "foo@bar.com", recognized with a simple regular expression.

`LocalLocationItem`: Local file paths, e.g. "C:\folder\foo.txt", recognized with a simple regular expression.

`UncLocationItem`: Universal Naming Convention file paths, e.g. "\\server\folder\foo.txt", recognized with a simple regular expression.

`IdentifierItem`: Uses a simple regular expression to find the kinds of names that are often used for software-related artifacts, e.g. "FooBar", "foo_bar", "foo123", etc..

`NumberItem`: Uses a simple regular expression to find strings of digits, e.g. "12345", because software-related artifacts are often numbered.

`HttpLocationItem`: While URLs are detected using a regular expression, redirection can cause a single page to have multiple URLs so the `HttpLocationItem` attempts to fetch any URL found by the regular expression, and uses the final URL to create the identifier.

`BugItem`: At Microsoft (and likely elsewhere), people use a wide variety of wording to reference bugs ("bug 12345", "resolves 12345", "duplicate of 12345", "fixes 12345, 23456, and 34567", etc.) the regular expression used by the `BugItem` scanner is much more complex than the others. At Microsoft there are hundreds of bug databases, with conflicting number spaces, so more work must be done to resolve the reference to a specific database. Most allusions to bugs rely on context to imply the particular database. The current system resolves the ambiguity by querying the index to find the bug database that's most strongly connected to the item that includes the scanned text. When a candidate bug database is detected, the `BugItem` scanner queries it for the specified bug number and then discards reference if the bug don't exist. (Note that we make no attempt to resolve vague allusions like, "that bug we worked on yesterday".)

For example, consider a hypothetical check-in message: "This fixes bug 12345, which was an off-by-one error causing an array scan to terminate before the end. It also caused that intermittent problem reported by foo@bar.com." The message contains a reference to a bug and a reference to an email address. When the `BugItem` scanner detects the bug reference, it creates an instance of `BugItem` and an instance of `MentionsLink` associating it with the present `CheckInItem`. The `NumberItem` scanner also detects a reference to the number 12345, and therefore instantiates a `NumberItem` instance and a `MentionsLink`. A similar process happens with the `EmailAddressItem` scanner. The other scanners are run but don't detect any allusions. Thus the knowledge casually coded into the check-in message is normalized into data structures.

In the future we expect to implement scanners for build numbers, knowledge base articles, domain account aliases, and method names. Method names might be approached with a combination of dictionary-based and regular expression techniques but both are problematic because, at least in current work practice, people often accidentally misspell identifiers and intentionally transform them into plurals (-*s*) and gerunds (-*ing*).

3.4 More about Provisioning

The crawlers, crackers, and scanners work in concert to provision the index with artifacts and relationships. They may be augmented by other techniques. Text similarity, used by the Hipikat system [1], could be applied to the index to create additional links, using the `Link` confidence property to represent the degree of similarity. Relationships between items discovered in navigation history, used by the Team Tracks system [2], could be turned into additional links (again using the confidence property), and indeed be extended beyond methods to include other artifacts such as bugs, emails, specs, URLs, etc. Simple rule-based approaches could be used to associate check-ins with contemporaneous bug actions by the same author. There may be other automated techniques for provisioning the index. They would work independently but the combined effect creates a richly-connected graph of software-related artifacts. In addition to automated techniques tools could allow the user to create items and relationships, such as annotations, and user-specified keywords that are automatically linked to the items that contain them.

Note that in several cases the crawlers, crackers, and scanners will submit items that may already be in the index. For example the bug database crawler may create a `BugItem` for bug 12345 and

Table 2: The number of links of each type.

Type	Count
Contains	5,578,988
Mentions	1,864,132
Next	1,590,770
Author	470,569
Recipient	49,118
Owner	43,683
Resolved-by	6,426
Closed-by	5,801
Reply	3,911

Table 3: The number of textual allusions by type of the item in which they were found and the type of item referred to.

		Mentioning Item Type				
		SCCS check-in	Email	Bug revision	Bug	**Total**
Mentioned Item Type	Identifier	221,510	197,301	298,173	224,436	941,420
	Number	309,403	209,630	89,139	66,431	674,603
	Bug	81,680	20,440	3,563	1,826	107,509
	HTTP URL	939	22,542	28,748	25,480	77,709
	Local file path	495	35,729	10,164	6,458	52,846
	Server file path	280	4,272	2,359	1,892	8,803
	Email address	21	773	241	207	1,242
	Total	614,328	490,687	432,387	326,730	1,864,132

the `BugItem` scanner may do the same. While the crawler has detailed information about the bug, and may thus populate the optional properties, the scanner knows only enough to create the item's identifier. To further complicate matters, either may encounter the bug first. The semantics of submission described in the section 2 allow the crawler and scanner (and any other mechanisms that provision the index) to operate independently.

4. RESULTS

We have built an index based on some of the data sources related to the development of the Microsoft Windows operating system. Activity between July 1st, 2005 and January 31st, 2006 has been gathered from eighteen source code control system databases one bug database. (For this analysis the contents of the source code control system file revisions were not gathered.) In addition, the index includes about twelve thousand emails dating from 2005 from several internal build-related email discussion lists.

The index includes 4,734,565 items and 9,613,398 links of various types (see Tables 1 and 2). (Note that each bug is composed of a series of *bug revisions*, each representing a specific action done to the bug: create, edit, resolve, close, etc.) While the average node degree (i.e. the average number of edges incident to the node) is 2.0, the distribution is highly skewed. The degree varies greatly by the node type, as shown in the *Avg. Degree* column of Table 1.

The links representing textual allusions are plentiful, comprising 19% of the links in the index. Table 3 categorizes the `MentionsLink` instances in the index by the type of item in which the allusion occurred and the type of item alluded to. (Note that text associated with a bug is counted twice, once for the bug revision and once for the bug itself; the *Bug revision* and *Bug* columns in Table 3 should be interpreted accordingly.)

5. DISCUSSION AND CONCLUSION

This effort combines the traditional representation of structured relationships with detection of textual allusions. These allusions contribute a substantial portion of the relationships represented in the index. Textual allusions are only one way to go beyond structured relationships. Text similarity, explored in the Hipikat project, and navigation paths, explored in the Team Tracks project, and other techniques, may complement the structured and allusive relationships.

The use of identifiers and the associated semantics of submitting items to the index combine to support decoupling of the various components contributing to the index. This is an important property if the system is to be allowed to grow organically, allowing new data sources to be added without much concern to the prior or subsequent additions.

While this initial work suggests that the approach is promising, there is a lot of work to do to evaluate whether it has benefits in real-world usage of tools built on the index. Tools that employ the index to deliver benefit to developers need to be fleshed out and evaluated in lab-based and field studies. Once deployed, their utility in helping developers understand the rationale behind code will become clearer.

If such a system is useful for software developers and their cohorts then it may be applicable to other knowledge work environments.

6. REFERENCES

[1] Davor Čubranić, Gail C. Murphy, Janice Singer, Kellogg S. Booth, "Hipikat: A Project Memory for Software Development," in *IEEE Transactions on Software Engineering* 31(6), IEEE Computer Society, pp. 446-465, June, 2005.

[2] Robert DeLine, Mary Czerwinski, and George Robertson, "Easing Program Comprehension by Sharing Navigation Data," in *Proc. VL/HCC'05*, IEEE Computer Society, pp. 233-240, 2005.

[3] Jon Kleinberg, "Authoritative Sources in a Hyperlinked Environment," in *J-ACM* 46(5), pp. 604-622, 1999.

[4] Thomas D. LaToza, Gina Venolia, Robert DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," to appear in *Proc. ICSE'06*, ACM Press, 2006.

[5] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford Digital Libraries Working Paper, 1998.