Traceclipse: An Eclipse Plug-in for Traceability Link Recovery and Management

Samuel Klock, Malcom Gethers, Bogdan Dit, Denys Poshyvanyk
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185
{skkloc,mgethers,bdit,denys}@cs.wm.edu

ABSTRACT

Traceability link recovery is an active research area in software engineering with a number of open research questions and challenges, due to the substantial costs and challenges associated with software maintenance. We propose Traceclipse, an Eclipse plug-in that integrates some similar characteristics of traceability link recovery techniques in one easy-to-use suite. The tool enables software developers to specify, view, and manipulate traceability links within Eclipse and it provides an API through which recovery techniques may be added, specified, and run within an integrated development environment. The paper also presents initial case studies aimed at evaluating the proposed plug-in.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation*

General Terms

Documentation; Management.

Keywords

Traceability, information retrieval.

1. INTRODUCTION

The maintenance phase of a software project's lifecycle presents developers with numerous expensive challenges. One challenge receiving attention from the research community is the recovery of traceability links between software artifacts. As software systems evolve, there is a corresponding evolution of links between various software artifacts but developers often ignore to keep links up-to-date. Failing to maintain such links makes it difficult to identify relationships between various types of artifacts, such as between high-level artifacts (e.g., requirements) and low-level artifacts (e.g., source code). Traceability link recovery techniques have been proposed to identify such links.

TEFSE'11, May 23, 2011, Waikiki, Honolulu, HI, USA

The contribution of this paper is a tool for traceability link recovery and management that supports existing techniques described elsewhere in the literature. We introduce Traceclipse, a plug-in for Eclipse IDE that provides a user-friendly interface to traceability link recovery techniques and enables developers to manage traceability links (i.e., accept or reject those discovered automatically, manually specify links, and store links) in an intuitive and easy way.

2. TRACEABILITY LINKS RECOVERY USING INFORMATION RETRIEVAL

Traceclipse is intended to provide support mainly for IR-based traceability link recovery techniques, although support may be added without significant difficulty for other retrospective techniques or prospective techniques. Here, we briefly discuss traceability links recovery techniques based on information retrieval.

The basic steps take the following form. First, a corpus for the target artifacts (i.e., set of software artifacts being traced onto) is constructed. After construction, preprocessing (i.e., the removal of non-literals, the splitting of identifiers, the removal of stop words, etc.) is applied.

An information retrieval technique is then applied to index the corpus and each element of the source artifacts (i.e., set of artifacts being traced) is converted into a query that is then compared to each document in the corpus. In a typical usage scenario, a list of the most similar pairings of source and target artifacts would then be presented to the user for inspection and validation.

Traceclipse implements a preprocessor for Java artifacts and a basic IR-based recovery technique that uses Lucene[1].

3. PLUG-IN FUNCTIONALITY

Traceclipse, which is available for download[2], provides services in two general areas: *links management* and *links recovery*.

3.1 Management

Traceclipse provides basic functionality for traceability links management. The plug-in maintains a set of traceability links for each Eclipse project; each link identifies a source artifact and an associated target artifact, along with a similarity score computed between them. Each project is also associated with a directory containing documentation artifacts.

[1]http://lucene.apache.org/
[2]http://www.cs.wm.edu/semeru/traceclipse

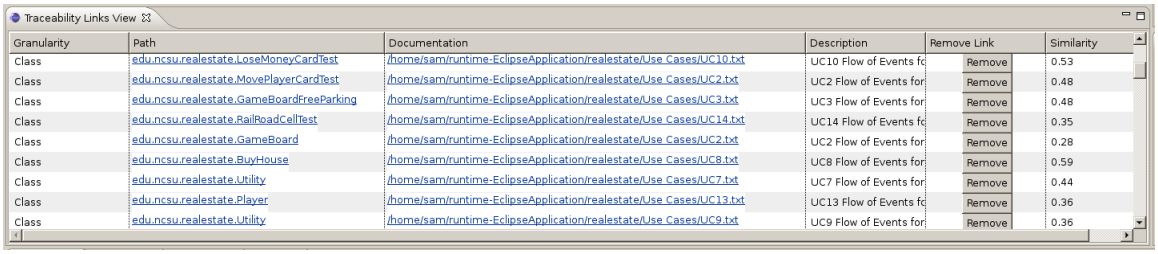

Figure 1: The Traceability Links View.

Within a project, a user may view the links specified for a particular source artifact or for the entire project. The user may also manually specify traceability links for pairs of artifacts. Traceclipse currently recognizes implementation artifacts at the package, file, class/interface, and method levels of granularity. To add a link for a particular artifact, the user may right-click it and select "Traceability->Add link..." from the artifact's context menu. A dialog box appears showing both a list of possible target artifacts and, for textual artifacts, a preview of the selected artifact's contents. When the user clicks "OK", a traceability link for the selected pair of artifacts is added to the project's set, and its similarity is set to "User-specified", as a way of marking a strong connection between those artifacts.

The user may view the traceability links specified for a given artifact (or for the entire project) by right-clicking the artifact and selecting "Traceability->View links...". A new view implemented with this plug-in called the *Traceability Links View* is populated with links associated with the selected artifact. If the links are viewed for an entire project, the view is populated with all the links specified for the artifacts in that project.

The Traceability Links View (see Fig. 1) shows links in terms of the granularity of implementation artifacts, the names of the implementation artifacts, the names of the documentation artifacts, a description of the documentation artifacts, and a similarity score between the implementation and documentation artifacts. If the artifacts for a given link are user-specified, then the similarity column shows the text "User-specified." Clicking on the name of either the implementation or the documentation artifact opens the appropriate Eclipse editor for viewing that artifact. Each row in the view also provides a "Remove" button that, when clicked, removes the associated link from the project's set.

For convenience of storage and analysis, Traceclipse stores the traceability links for a given project in XML format. The plug-in's internal representation of traceability links are serialized into XML using XStream[3] and are stored under the associated project's directory in .trace/links.xml (e.g., for a project in the workspace named "foo", the links are stored in [workspace_directory]/foo/.trace/links.xml). The links file may be easily parsed for analysis outside of Eclipse and without the assistance of Traceclipse's API.

3.2 Recovery

For semi-automatic retrospective techniques, Traceclipse provides a new wizard-based interface from which these techniques may be parameterized and run. Via the interface, a user selects a level of granularity with which to view source code artifacts, preprocessing options for source and documentation corpora, a traceability recovery technique and associated parameters, a directory containing documentation to which to trace, and criteria for proposed links to view. After the wizard closes, the selected technique is run and candidate links meeting the user's criteria are presented in the Traceability Links View for inspection.

The wizard presents standard preprocessing options to the user. Supported options include the removal of non-literal characters, the splitting of identifier names (using either the camel case or underscore conventions), word stemming (either for all words or nouns only), and the removal of stop words. For the latter option, the user is expected to supply a text file containing stop words to remove with one word per line. The Porter stemmer[4] is used to stem words. Unfortunately, though the option is given to do so in the link generation wizard, the preprocessor currently is not capable of distinguishing between nouns and other parts of speech. Comments for all code are included at each level of granularity and Javadocs for methods are included when the preprocessor is run at the method-level granularity.

Preprocessing is only completed after the user clicks the wizard's "Finish" button, and its output is stored in the project's directory under .trace/ppsource (for implementation artifacts) and .trace/ppdoc (for documentation artifacts). Each document in each corpus is assigned a name of the form [type][id].txt, where [type] is either "doc" (for documentation artifacts) or "source" (for implementation artifacts) and [id] is a unique integer. These files are mapped to the original artifacts to which preprocessing is applied via a file called "mapper.txt" that is stored in .trace/ppdoc and .trace/ppsource respectively.

The documentation located in the user-selected directory should be stored in plain-text format. Currently this format is the only one supported, but in the future we plan to allow users to provide their documentation in other formats. For best results, the user should verify that documentation is partitioned into traceable *chunks* (i.e., each requirement should reside in its own file, each use case should reside in its own file, etc.). The user may decide to trace a query onto multiple different kinds of documentation artifacts simultaneously if desired, but for precision it is advisable to focus on only one kind of artifact at a time.

Each technique included in the wizard specifies its own parameters to present to the user. The example included with Traceclipse, which uses Lucene, allows the user to specify a maximum length for fields associated with each document in

[3]http://xstream.codehaus.org/

[4]http://tartarus.org/ martin/PorterStemmer/

the documentation corpus; smaller values improve speed and reduce the technique's memory footprint, but they constrain the amount of information extracted from each document to a certain size. In practice, if documentation artifacts tend to be small in size, then small field lengths should be acceptable. Other techniques implemented may specify their own parameters (e.g., for Latent Semantic Indexing (LSI) [9], a value for the dimensionality reduction factor).

The criteria for proposed links can take one of two forms: a fixed number of links or a threshold for artifact pair similarities displayed. In the first case, the user selects a number, k, indicating the number of candidate links, with the highest similarity scores, to provide to the user. In the second case, the user selects a threshold p for the similarity score, and the technique returns only those links with a similarity score above that threshold.

Upon completion, the Traceability Links view displays traceability links, according to the specified criteria, and the user has the ability to remove invalid links.

4. EXAMPLE USAGE

We ran Traceclipse in the context of several projects to test and assess its functionality. Two of the projects, iTrust and RealEstate, are available via the Repository for Open Software[5] (ROSE). CM1 is a science instrument developed by NASA and EasyClinic is a system used to manage a doctor office's appointments developed by students. The latter two systems are used in the TEFSE Challenge[6].

Our plug-in required at most a few minutes of computation time with some management functions completed almost instantaneously. Traceclipse appears to cope well with compilation errors in code (e.g., syntax errors, unresolvable package references) and in our tests the tool did not encounter any constraints imposed by the Java heap space. One concern is that the Traceability Links View can take a significant amount of time (i.e., several seconds) to render if more than a handful of links are inserted into it. We believe this is due to the fact that there are several widgets embedded in the view for each traceability link, which makes rendering the table more difficult to do. A possible solution to the problem is to replace the widgets from the view with context menus providing the same functionality.

Table 1 summarizes our experience with the ROSE projects using traceability link generation at method-level granularity with all preprocessing options except for stop word removal enabled. These tests were run on a machine with a dual-core processor clocked at 2.8 GHz and six GB of RAM running Ubuntu 10.04. Neither project is particularly large, but they both give some insight on how Traceclipse might perform in real-world contexts. The corpora produced by the preprocessor did not exceed a megabyte in size for either project. Neither project required a substantial amount of time to produce the requested set of traceability links, although the amount of time required for iTrust suggests that projects of an order of magnitude larger may require an hour or more to process. Most of the time required in the case of iTrust was consumed by Lucene; the preprocessor finished in well under a minute. More efficient recovery techniques may require less time to run.

[5]http://agile.csc.ncsu.edu/rose/

[6]http://www.cs.wm.edu/semeru/tefse2011/Challenge.htm

	RealEstate	iTrust
Lines of code	$< 1,000$	About $7,700$
Size of artifacts	847 KB	2.0 MB
Documentation types	Use cases	Uses cases, non-functional requirements
Size of corpora	115 KB	942 KB
Size of Lucene index	6 KB	37 KB
Time required	Seconds	Five minutes
Links requested	500	500
Size of links file on disk	335 KB	357 KB

Table 1: Projects used to assess Traceclipse's traceability links generation.

Project Traced	*Correct*	*Precision*	*Recall*
EasyClinic (UC)	26	0.385	0.108
EasyClinic (ID)	71	0.465	0.478
EasyClinic (TC)	70	0.400	0.269
CM1	75	0.240	0.400

Table 2: Precision-recall performance of the Lucene technique at cut point 75.

Note that this section is *not* intended to constitute a scientific evaluation of the tool for management or research purposes. We leave that as future work.

4.1 Lucene Performance

As a proof-of-concept, the precision-recall performance of the Lucene recovery technique is shown in Table 2. We applied Lucene to each dataset using standard preprocessing options (i.e., word stemming, removal of non-literals, and identifier splitting), and up to 75 traceability links with similarity scores of at least 0.15 were selected. We manually assessed if the proposed traceability links returned by Traceclipse are correct or not.

5. PLUG-IN DESIGN

A central goal for Traceclipse's design is that it should be easily *expandable*. While we intend for Traceclipse to facilitate routine traceability links-related tasks (e.g., specifying and viewing links), we also want the tool to be a suitable platform on which to conduct research on future traceability link recovery techniques. Accordingly, the tool should provide a straightforward API upon which to build and test new techniques.

To meet that goal, we developed a simple API to implement new techniques. Retrospective (particularly IR-based) techniques may be added in the context of the recovery wizard by implementing a pair of interfaces and prospective techniques essentially need only to interact with Traceclipse's manager module. The relevant elements of the design are described in the following subsections and procedures for adding new techniques are described in the next section.

5.1 Traceability Links

The package in which the concepts of a traceability link and a set thereof is specified in Table 3. There, two classes and an enumerated type are defined. The class `TraceabilityLink` represents a traceability link in terms of a implementation artifact, a documentation artifact, a similarity score, a level of granularity, and an optional textual description. Objects of the type are represented in aggregate via the class `TraceabilityLinkSet`. Levels of granularity are defined in the enumerated type `Granularity`.

Element	*Packages*	*Description*
Traceability Links	`traceability.links`	API for interacting with traceability links
Manager	`traceability.management`	API for managing traceability links within a project
Recovery	`traceability.recovery` `traceability.wizard`	API for building new recovery techniques

Interface	`traceability.dialog`
	`traceability.popup.actions`
	`traceability.views`
		Elements of Traceclipse's user interface; the traceability links view; etc.

Miscellaneous	`traceability` `traceability.util`	Miscellaneous/utility class definitions

Table 3: Packages for various elements of the design.

Instances of `TraceabilityLinkSet` are serialized into XML for the purpose of storing traceability links. That functionality is described in the next subsection.

5.2 Manager

The abstraction of traceability links management is handled by the class `Manager` in the package given by Table 3. The `Manager` class constitutes the foundation of Traceclipse, as it provides an interface between the other elements of the tool (e.g., the UI and recovery techniques) and the traceability links set for each project in the Eclipse workspace.

`Manager` is a singleton class and its responsibilities include creating empty sets of traceability links for each project, adding links to and removing links from those sets, and extracting links from the sets for analysis. The `Manager` also keeps track of where the sets are stored.

The `Manager` handles the information about a single project at a time. To interact with the traceability links for a particular project, interested entities must first have the `Manager` change its current project to the desired project. The `Manager` will save the set of traceability links for the current project to disk and load the set for the desired project, whereupon interested artifacts may use the `Manager` to view and manipulate the contents of the set. From the perspective of a user, existing elements of the UI set the current project appropriately in response to user actions; additions to the UI will have to take this into account.

5.3 Recovery

The relevant packages for recovery-related code are given in Table 3. In the `recovery` package, Traceclipse provides a preprocessor for Java artifacts, along with two interfaces and an enumerated type defining contracts for recovery techniques. A sample implementation using Lucene also resides in that package.

The user is expected to activate (retrospective) recovery techniques via the wizard interface (see Section 3.2). The wizard is defined using Eclipse's wizard API (`org.eclipse.jface.wizard`) and is divided into a series of pages corresponding to the steps described earlier. The recovery wizard runs the preprocessor and the user-specified recovery technique (after the user presses the "Finish" button) and is responsible for collecting links from the technique, adding them to the current set via the `Manager`, and making the set available to the Traceability Links View (see Figure 1).

5.3.1 Eclipse JDT API

The preprocessor (`JavaDocumentPreprocessor`) relies heavily on the API from Eclipse's Java Development Toolkit (JDT). When generating documents at the class- and method-levels of granularity, the preprocessor uses the API to build abstract syntax trees from the source code to determine where classes and methods begin and end. In addition, the API is used to tokenize the source code to facilitate filtering non-literals and Java keywords from input documents.

The crucial elements of the JDT used to build the source corpus reside in `org.eclipse.jdt.core.compiler` and `org.eclipse.jdt.core.dom`. In the former package, the `Scanner` class is used to tokenize Java source, enabling the preprocessor to efficiently ignore Java keywords and, where appropriate, non-literals. The `Scanner` is applied to source artifacts at all levels of granularity, but it is not applied to documentation artifacts.

In the latter package, Traceclipse uses several classes to produce and analyze abstract syntax trees. One of the most important classes is `ASTParser`, which is used to build the abstract syntax trees. Various other classes, such as `AST` and `ASTNode`, are used to traverse the trees and extract interesting nodes for the purposes of preprocessing (e.g., `CompilationUnit`s, `Type`s, and `MethodDeclaration`s, depending on the level of granularity). Information extracted from these nodes is used to guide preprocessing. In each case, the JDT's representation marks where the unit begins and ends in the source file, which helps the preprocessor divide source files into documents prior to further preprocessing.

5.4 Interface

Traceclipse's provides a new view for Eclipse, called Traceability Links View that is implemented in `traceability.views`. It also provides functionality to context menus for various resources in Eclipse. These elements include `IProject` and `IFile` resources (as defined in Eclipse's resources API under `org.eclipse.core.resources`), along with `IType`, `IPackageFragment`, and `IMethod` (as defined in the JDT API under `org.eclipse.jdt.core`).

The extensions add a submenu for each resource named "Traceability". For each resource apart from `IProject`, the menu contains two items, "Add link..." and "View link...", with the obvious semantics. For `IProject` resources, the menu also contains two items, "View links..." and "Build project corpus...". The first item has predictable semantics; the second item brings up the recovery wizard.

The classes responsible for handling events generated by the new menu items are located in `traceability.popup.actions`. Their names give their expected behavior; for example, `AddLink` is used to create manually specified traceability links. `ViewLinks` provides one of two ways for the user to populate the Traceability Links View with links; the other way is via link generation through the recovery wizard.

6. EXTENDING TRACECLIPSE

Traceclipse is designed to be easily extended with new recovery techniques. While the API is designed primarily

to support retrospective techniques (particularly those that are IR-based, given the supplied preprocessing functionality), it may also be used to develop and deploy prospective techniques. We describe how to do so in the following two subsections.

6.1 Retrospective Techniques

Developers who wish to contribute retrospective techniques to Traceclipse must follow two basic steps. First, they must implement the interfaces `RecoveryTechnique` and `RecoveryTechniqueParams` in `traceability.recovery` (see Figure 2). Second, they must add a line of code to `TechniqueWizardPage` in `traceability.wizard` to get the wizard to recognize the new technique.

For the first step, developers must implement an interface for the technique itself via `RecoveryTechnique`. The interface specifies four methods, which respectively specify the technique's name, its parameters, a way of running it, and a way of obtaining links from it following its execution. The parameters for the technique are specified via a second interface, called `RecoveryTechniqueParams`. All techniques are required to implement this interface and the parameter method for each technique (`RecoveryTechnique.getParameters`) is expected to return an instance of `RecoveryTechniqueParams` that will be used by the technique.

The `RecoveryTechniqueParams` interface specifies several methods used to parameterize recovery techniques. `RecoveryTechniqueParams` is expected to function as a mapping from parameter names (Strings) to values for those parameters. Each instance is required to provide a method for setting a special parameter, which is the granularity with which to view source artifacts. Otherwise, the set of parameters (if any) is decided entirely by the technique. The interface requires implementers to provide labels for each parameter, to specify types for each parameter, and to provide ways of getting and setting the value of each parameter. Legal types are defined in the enumerated type `ParameterType` in `traceability.recovery`; they are used by the wizard to build controls allowing the user to set values for the parameters. Currently, supported types include text, booleans (check-boxes and radio buttons), integers, and reals. Contributing additional parameter types will require modifying `ParameterType` to include them and `TechniqueWizardPage` to generate appropriate controls. (This functionality may be transferred to the enumerated type, if deemed appropriate.) Classes implementing `RecoveryTechniqueParams` should provide default values for each parameter; these are used to populate appropriate fields in the wizard.

Apart from implementing the interfaces, developers must also add a line to `TechniqueWizardPage` in order for their techniques to appear in the wizard. `TechniqueWizardPage` has as a member an array of `RecoveryTechnique`s called `TECHNIQUES`; developers must create an instance of their new technique in that array for it to appear in the wizard. The wizard itself will handle the generation of controls for the technique. Again, this is a requirement that may be relatively easy to obviate; see Section 7.

6.2 Prospective Techniques

The tool provides no direct support for prospective traceability apart from allowing users to manually specify and remove links. Automated prospective traceability techniques may rely on Traceclipse to manage links via the `Manager` class. For example, a simple prospective technique might keep track of the source artifacts on which a developer is working (and, by extension, the project on which the developer is working). When a developer opens a documentation artifact, the technique might automatically prompt the `Manager` to add a link between the source and documentation artifacts. More sophisticated prospective techniques may follow this basic form. Thus, Traceclipse does provide a useful, if modest, API for contributing prospective techniques.

```
public interface RecoveryTechnique {
     public String getName();
     public RecoveryTechniqueParams getParameters();
     public boolean run(IProject project);
     public Set<TraceabilityLink> getRecoveredLinks();
}

public interface RecoveryTechniqueParams {
     public String [] getParameterIDs();
     public String getLabelForParameter(String id);
     public ParameterType getTypeForParameter(String id);
     public void setValueForParameter(String id, Object val);
     public Object getParameterValue(String id);
     public void setGranularity(Granularity granularity);
}
```

Figure 2: Interfaces for retrospective techniques.

7. FUTURE WORK ON TRACECLIPSE

Although we provide a working plug-in for researchers and practitioners for traceability link recovery, there are features which will be further developed.

One goal for future development is to enable developers to do so by having them specify the location of class files for their new techniques in a configuration file, which is then read and used in the context of Java's reflection API to pull the technique into the recovery wizard. Ideally, developers should be able to implement new techniques and then *configure* Traceclipse to use them (i.e., via a settings file in a well-known location). A second goal is to develop project-specific preferences for Traceclipse. Traceclipse currently does not support preferences of any kind. However, the user should ideally be able to control, for example, where sets of traceability links are stored on disk.

Another avenue for future work is extending Traceclipse to support languages other than Java. The current preprocessor defined in `traceability.recovery` relies heavily on the JDT API, which means that adding new language support will require either substantially modifying it or adding a new preprocessor (e.g., Eclipse CDT API for C programs).

There are avenues for future work on Traceclipse, which mainly deal with extending Traceclipse to support more features (i.e., languages other than Java) and improving its existing functionality, such as the efficiency of its preprocessor. More important, it would be wise to assess how effective the tool is as a support both for real-world software projects and as a research platform.

8. RELATED WORK

Research in the field has dealt with three means of traceability link recovery: manual, automatic, and semi-automatic [21]. Manual methods entail that developers or analyst track links by hand, generally during the implementation process; the analyst is responsible for finding and evaluating links

without the assistance of automated techniques. Fully automatic techniques entail the use of specialized software that tracks the creation and evolution of software artifacts and that makes final decisions on links itself.

Most work has occurred in the field of semi-automatic traceability link recovery, in which both the software tool and the analyst play a role. The focus of this project is the development of a semi-automatic tool, hence the focus of this section will be on semi-automatic traceability links recovery, with an emphasis on the use of IR techniques in candidate link generation.

8.1 Semi-automatic Traceability Links Recovery

Antoniol et al. produced early work in (semi-)automatic traceability link recovery [3]. Their work focused on the selection of properties along which artifacts may be matched (e.g., class names, methods, fields, attributes; developer-produced dictionaries; etc.). Prior to their work, automated traceability techniques were not well-studied. Antoniol et al. point to, for example, the development of an annotation language for Ada [16].

Since their work, researchers have developed a number of approaches to semi-automatic traceability link recovery. Qusef et al. divide these into three categories: heuristic-based, data-mining based, and IR-based [20]. Their work is an example of heuristic-based traceability link recovery. To discover traceability links between JUnit test cases and tested code, the authors applied a heuristic: any class that affects the outcome of the last *assert* statement in a test case is a class under test. Egyed and Grunbacher [10] proposed pairing requirements with execution traces on the assumption that source code artifacts exercised in those traces implement the requirements. Data-mining techniques have been used to find traceability links among source code artifacts, which is useful for detecting relationships among classes [20]. IR-based techniques are discussed more fully in the following subsection.

Semi-automatic traceability link recovery relies on two actors to produce useful results: the automated tool (responsible for finding candidate links) and the analyst (responsible for evaluating and accepting/rejecting candidate links). In most research, the focus is on describing an automated technique that produces the best results (i.e., proposes the most correct traceability links with as few incorrect links as possible). Hayes and Dekhtyar [13] point out that, in addition to recall and precision, such techniques ought to focus on usability and earning the confidence of the analyst to maximize the effectiveness of the human factor; analysts who find tools difficult to use or do not trust them to yield good results tend to use them less effectively. One of the goals of this project is to produce a tool that is both effective and highly usable.

Conversely, a related issue in the effectiveness of semi-automatic traceability techniques, as well as, presumably, fully automatic techniques, is developer discipline. Many techniques, especially those that rely on textual information (i.e., IR), rely on developers to follow good programming and documentation conventions. Cleland-Huang et al. [7], for example, advise developers to meaningfully organize documentation; state requirements clearly, concisely, and meaningfully; and constrain the language used in documentation to a small, domain-specific vocabulary. Similarly, De Lucia et al. [15] noted the problem of poor programming conventions, for example, poorly selected names for identifiers. They suggested an IR-based traceability system called COCONUT to help programmers wisely select identifier names based on the artifacts related to their task. More recently, the tool has been extended, resulting in CodeTopics [11], which utilizes topic models to show developers the similarities between source code and high-level artifacts as well as providing other useful functionality.

8.2 Traceability using Information Retrieval

IR-based traceability recovery techniques rely on textual information to draw inferences about how software artifacts are related. Typically, artifacts are partitioned into sets of documents called corpora. Preprocessing techniques, such as word stemming and identifier splitting, are then applied to the documents in each corpora, yielding a series of terms presumed to be meaningful in each document. An IR-based algorithm is then used to compute similarity scores between artifacts from each corpus, which are then used to propose candidates for links [7].

Antoniol et al. introduced the usage of IR techniques to the problem of traceability [2]. In particular, their work evaluated the effectiveness of vector-space and probabilistic network models as similarity metrics in semi-automated links recovery. They found both methods promising.

Subsequent work expanded on the IR paradigm that Antoniol et al. established. Marcus et al. [17] in particular expanded on the use of Vector Space Models (VSM) for traceability link recovery, using LSI; they claimed to obtain results at least as good as Antoniol et al. with less preprocessing required and language-independence. Others later expanded on their investigation; De Lucia et al. [14], for example, integrated LSI-based traceability link recovery into an artifact management system and assessed its effectiveness in terms of usability, efficiency, and accuracy with reasonably promising results. As another example, Abadi et al. [1] compared a number of different IR techniques, including LSI, variants thereof, and a new technique using the Jensen-Shannon similarity model. They found that the vector-space and Jensen-Shannon (JS) similarity models were most effective for traceability link recovery. Capobianco et al. [6] proposed a technique using the B-spline method, which is based on numerical analysis. The authors also claimed to obtain superior results to both vector-space models and LSI. More recently, Oliveto et al. [18] demonstrate that across various systems LSI, JS, and VSM result in equivalent performance while recovery techniques based on topic models yield results orthogonal to the aforementioned techniques. One potential issue encountered when applying information retrieval techniques to recover links between various types of artifacts relates to vocabulary mismatch. Cleland-Huang et al. [8] and Gibiec et al. [12] propose combining webmining and information retrieval techniques to alleviate the issue and improve their accuracy.

8.3 Traceability Tools

A number of tools for traceability link recovery and management exist in the literature. An early example is TOOR, developed by Pinheiro and Goguen in 1996 [19]. It enables the specification and management of links among multiple kinds of artifacts using user-defined relations. The latter feature is intended to make the intended meaning of traceability

links easier for developers to understand. TOOR predates most of the research on semi-automatic recovery techniques, hence it provides no automated recovery feature.

Asuncion et al. provides a pair of more recent examples [4, 5]. The first of these examples was developed around 2007 with support from the Wonderware commercial software company. The unnamed tool is intended to support traceability links management with a focus on minimizing overhead and enabling reuse.

The second was described in 2010; it is a tool given in three components intended to support prospective traceability link recovery using topic modeling as a guide for finding new links and paring invalid ones. Notably, neither of these tools rely on traditional IR-based techniques described in the literature. The first relies on manual management and the second relies primarily on automated prospective traceability. An example more consistent with the literature described here is given by De Lucia et al. in the form of ADAMS, a tool for traceability link recovery and management [14]. The tool relies on LSI to retrieve candidate links, assess their probable quality, and make suggestions to developers during software evolution.

It is worth noting that no tool in the literature appears to serve as a platform for general research in traceability link recovery. In this respect, Traceclipse is a novel contribution.

9. CONCLUSION

In this paper, we presented a new plug-in for traceability link recovery and management. Traceclipse provides support for manual specification and management of traceability links and it supplies an API which new techniques for traceability link recovery may be built and analyzed. It stores traceability links in a convenient, portable XML file, and provides functionality for viewing/assessing traceability links. Accordingly, we hope Traceclipse may be used both as a research platform and as a practical system for traceability link recovery in real-world development projects.

Acknowlegemetns

This work is supported by NSF CNS-0959924 grant. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

10. REFERENCES

[1] A. Abadi, M. Nisenson, and Y. Simionovici. A traceability technique for specifications. *Proc. of 16th ICPC*, pages 103–112, 2008.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE TSE*, 28(10):970–983, 2002.

[3] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability recovery: selecting the basic linkage properties. *Sci. Comput. Program.*, 40(2-3):213–234, 2001.

[4] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *Proc. of 32nd ICSE*, pages 95–104, 2010.

[5] H. U. Asuncion, F. François, and R. N. Taylor. An end-to-end industrial software traceability tool. In *Proc. of 6th ESEC/FSE*, pages 115–124, 2007.

[6] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella. Traceability recovery using numerical analysis. In *Proc. of 16th WCRE*, pages 195–204, 2009.

[7] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. Best practices for automated traceability. *Computer*, 40(6):27–35, 2007.

[8] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker. A machine learning approach for tracing regulatory codes to product specific requirements. In *Proc. of 32nd ICSE*, pages 155–164, 2010.

[9] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[10] A. Egyed and P. Grunbacher. Automating requirements traceability: Beyond the record replay paradigm. In *Proc. of 17th ASE*, pages 163 – 171, 2002.

[11] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Codetopics: Which topic am i coding now? In *Proc. of 33rd ICSE, Formal Research Tool Demonstration*, to appear in 2011.

[12] M. Gibiec, A. Czauderna, and J. Cleland-Huang. Towards mining replacement queries for hard-to-retrieve traces. In *Proc. of the 25th ASE*, pages 245–254, 2010.

[13] J. H. Hayes and A. Dekhtyar. Humans in the traceability loop: can't live with 'em, can't live without 'em. In *Proc. of 3rd TEFSE*, pages 20–23, 2005.

[14] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM TOSEM.*, 16(4):13, 2007.

[15] A. D. Lucia, M. D. Penta, and R. Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE TSE*, 2010.

[16] D. C. Luckham, F. W. von Henke, B. Krieg-Brückner, and O. Owe. *ANNA: a language for annotating Ada programs*. Springer-Verlag, 1987.

[17] A. Marcus, J. I. Maletic, and A. Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(4):811–836, 2005.

[18] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proc. of 18th ICPC*, pages 68–71, 2010.

[19] F. A. Pinheiro and J. A. Goguen. An object-oriented tool for tracing requirements. *Software*, 13:52–64, 1996.

[20] A. Qusef, R. Oliveto, and A. D. Lucia. Recovering traceability links between unit tests and classes under test: An improved method. In *Proc. of the 26th ICSM*, 2010.

[21] S. Sundaram, J. Hayes, A. Dekhtyar, and E. Holbrook. Assessing traceability of software engineering artifacts. *Requirements Engineering*, 2010.