Empir Software Eng (2015) 20:1198–1236
DOI 10.1007/s10664-014-9339-3

Supporting and accelerating reproducible empirical research in software evolution and maintenance using TraceLab Component Library

Bogdan Dit · Evan Moritz · Mario Linares-Vásquez · Denys Poshyvanyk · Jane Cleland-Huang

Published online: 13 December 2014

Abstract Research studies in software maintenance are notoriously hard to reproduce due to lack of datasets, tools, implementation details (e.g., parameter values, environmental settings) and other factors. The progress in the field is hindered by the challenge of comparing new techniques against existing ones, as researchers have to devote a large portion of their resources to the tedious and error-prone process of reproducing previously introduced approaches. In this paper, we address the problem of experiment reproducibility in software maintenance and provide a long-term solution towards ensuring that future experiments will be reproducible and extensible. We conducted a preliminary mapping study of a number of representative maintenance techniques and approaches and implemented them as a set of experiments and a library of components that we make publicly available with TraceLab, called the Component Library. The goal of these experiments and components is to create a body of actionable knowledge that would (i) facilitate future research and (ii) allow the research community to contribute to it as well. In addition, to illustrate the process of using and adapting these techniques, we present an example of creating new techniques based on existing ones, in order to produce improved results.

Communicated by: Yann-Gaël Guéhéneuc and Tom Mens

B. Dit · E. Moritz · M. Linares-Vásquez · D. Poshyvanyk (✉)
The College of William and Mary, Williamsburg, VA, USA
e-mail: denys@cs.wm.edu

B. Dit
e-mail: bdit@cs.wm.edu
URL: http://www.cs.wm.edu/semeru/

E. Moritz
e-mail: eamoritz@cs.wm.edu

M. Linares-Vásquez
e-mail: mlinarev@cs.wm.edu

J. Cleland-Huang
DePaul University, Chicago, IL, USA
e-mail: jhuang@cs.depaul.edu
URL: http://facweb.cti.depaul.edu/jhuang/

Keywords Software maintenance · Reproducible · Experiments · Empirical case studies · TraceLab

1 Introduction

Research in software maintenance (SM) is primarily driven by empirical studies. Thus, advancing this field requires researchers not only to come up with new, more efficient and effective approaches that address SM problems, but most importantly, to compare their new approaches against existing ones in order to demonstrate that they are complementary or superior and under which scenarios. However, comparing an approach against existing ones is time consuming and error-prone for several reasons. For instance, existing approaches may be hard to reproduce because the datasets used in their evaluation, the tools and implementation, or the implementation details (e.g., specific parameter values, environmental factors) are not available (Dit et al. 2013d; Robles 2010; Mytkowicz et al. 2010; D'Ambros et al. 2012; Barr et al. 2010; González-Barahona and Robles 2012; Borg et al. 2013).

These problems are illustrated through a survey on feature location (FL) techniques by Dit et al. (Dit et al. 2013d), which revealed that only 5 % of the papers surveyed (i.e., three out of 60 papers) evaluated their approach using the same dataset used to evaluate other techniques, and that only 38 % of the papers surveyed (i.e., 23 out of 60 papers) compared their proposed feature location technique against any previously introduced feature location techniques. In addition, these findings are consistent with the ones from the study by Robles (2010), which determined that among the 154 research papers analyzed, only two made their datasets and implementation available, and the vast majority of the papers describe evaluations that cannot be reproduced, due to lack of data, details, and tools. Furthermore, a study by González-Barahona and Robles (2012) identified the factors affecting the reproducibility of results in empirical software engineering research and proposed a methodology for determining the reproducibility of a study. Similarly, Borg et al. (2013) conducted a mapping study investigating the relationship between evaluation criteria and results for traceability link recovery approaches based on information retrieval (IR). Their findings revealed that most studies were evaluated against datasets with fewer than 500 artifacts and, as a result, they identified the need for performing case studies on industrial-size datasets. They encouraged researchers to publicly provide the datasets and tools used in their evaluations and also provided a set of guidelines to raise the quality of publications in the field of software engineering research. In another study, Mytkowicz et al. (2010) investigated the influence of the omitted-variable bias (i.e., a bias in the results of an experiment caused by omitting important causal factors from the design) in compiler optimization evaluation. Their study showed that factors such as the environment size and the link order, which are often not reported and are not explained properly in the research papers, are very common, unpredictable, and can influence the results significantly. Moreover, D'Ambros et al. (2012) argued that many approaches in bug prediction have not been evaluated properly (i.e., they were either evaluated in isolation, or they were compared against a limited set of other approaches), and highlighted the difficultness of comparing results.

This issue of the reproducibility of experiments and approaches has been discussed and investigated in different areas of software maintenance research (Dit et al. 2013d; Robles 2010; Mytkowicz et al. 2010; D'Ambros et al. 2012; Barr et al. 2010; González-Barahona and

Robles 2012; Borg et al. 2013; Shull et al. 2008), and some initial steps have been taken towards solving this problem. For example, efforts for establishing datasets or benchmarks that can be used uniformly in evaluations have resulted in online benchmark repositories such as PROMISE (Menzies et al. 2012; Sayyad and Menzies 2005), Eclipse Bug Data (Zimmermann et al. 2007), SEMERU feature location dataset (Dit et al. 2013d), Bug Prediction Dataset (D'Ambros et al. 2012), SIR (Do et al. 2005), and others. In addition, different infrastructures for running experiments in SM and other fields were introduced, such as TraceLab (Cleland-Huang et al. 2011, 2012; Keenan et al. 2012), Rapid-I (2013), Mathworks (2013), Kepler (2013), and others. However, among these, a good candidate framework for facilitating and advancing research in software engineering and maintenance is TraceLab (see Section 3.2 for an in-depth comparison and discussion of TraceLab's features with other tools). More specifically, unlike the other frameworks, TraceLab is a plug-and-play framework that was *specifically designed for facilitating creating, evaluating, comparing, and sharing experiments* in software engineering and maintenance (see Section 3.1 for a detailed description of its features). These characteristics ensure that TraceLab makes experiments *reproducible*.

The goal of this paper is to ensure that a large portion of existing and future experiments in software maintenance research that are designed and implemented with TraceLab will be *reproducible*. To accomplish this, we analyzed the approaches presented in 27 SM research papers, identified their common building blocks, and we implemented them as components in a well-organized, structured, documented and comprehensive *Component Library* for TraceLab. In addition, we used the *Component Library* to assemble and replicate a subset of the existing SM techniques, and exemplified how these components and experiments could be used as starting points for creating new and reproducible experiments.

In summary, the contributions of our paper are as follows:

- a mapping study of techniques and approaches in SM (Section 4) to identify the set of techniques to reproduce as TraceLab experiments;
- a TraceLab *Component Library* (***CL***), which contains a comprehensive and representative set of TraceLab components designed to help instantiate the set of SM experiments, and a *Component Development Kit* (***CDK***), which serves as a base for extending this initial component base in order to facilitate the creation of new techniques and experiments;
- an example of reproducing a feature location technique using the proposed *CL,* and two examples of reproducing traceability link recovery techniques using the proposed *CL*; moreover, we illustrate two examples of using an existing technique as a starting point to design and evaluate new ideas;
- an online appendix that makes publicly available all the resources presented in this paper: https://www.cs.wm.edu/semeru/TraceLab_CDK/ and https://github.com/CoEST

The paper is organized as follows. Section 2 presents a motivating example that shows variability in results for applying a simple SM technique and the challenges of reproducing those results without complete details. Section 3 introduces background details about TraceLab and presents a comparison with other tools. Section 4 presents the mapping study performed, which we used to implement the Component Library and Development Kit (Section 5). Section 6 shows an example of reproducing two existing traceability link recovery techniques and an existing FL technique and presents alternatives on improving the FL technique. Finally, Section 7 discusses alternative uses of TraceLab and the CL. Section 8 presents some potential limitations, and Section 9 concludes the paper and introduces some ideas for future work.

2 Motivating Example

When new approaches are introduced, in general, authors rightfully focus more on describing the important details of the new techniques, and due to various reasons (e.g., space limitations) they may present only in passing the details of applying well-known and popular techniques (e.g., VSM), as they rely on the conventional wisdom and knowledge (or references to other papers for more details) about applying these techniques (Dit et al. 2013d; Robles 2010).

However, for a researcher who tries to reproduce the results exactly, it might be difficult to infer all the assumptions the original authors took for granted and did not explicitly state in the paper. Therefore, the reproducer's interpretation of applying the approach could have a significant impact on the results.

To illustrate this point with a concrete example, we applied the popular IR technique Vector Space Model (VSM) (Salton et al. 1975) on the EasyClinic system from TEFSE 2009[1] challenge to recover traceability links between use cases and class diagrams. We configured the VSM technique using four treatments consisting of all the possible combinations of two corpus preprocessing techniques and two VSM weighting schemes. The preprocessing techniques were *raw preprocessing* (i.e., only the special characters were removed) and *basic preprocessing* (i.e., special characters were removed, identifiers were split and stemmed). The weighting schemes used were *no weighting* and *term frequency-inverse document frequency* (*tf-idf*) weighting (Salton et al. 1975).

Figure 1 shows the precision and recall curves for recovering traceability links between use cases and classes on the EasyClinic dataset, using a VSM-based traceability technique and different preprocessing techniques (i.e., *raw preprocessing* in gray color and *basic preprocessing* in black color) and weighting schemes (i.e., *no weighting* in dash line and *tf-idf weighting* in solid line). The results in Fig. 1 show a high variety in the precision and recall values, based on the type of preprocessing and weighting schemes used. Assuming these details are not clearly specified in the paper, any of these configurations or variations of these configurations can be chosen while reproducing an experiment, potentially yielding completely unexpected and drastically different results. It is worth emphasizing that in our example we picked a small subset of the large number of weighting schemes and preprocessing techniques that can be found in the literature, and these options were deliberately picked to illustrate an example, as opposed to conducting a rigorous experiment to identify the configuration of factors that could produce the best results.

The main point of this example is that even in this simple scenario of using VSM for a typical traceability task, there are many options on how we can instantiate and use this technique, which leads to completely different results. However, all these problems could be eliminated if all these details are encoded in the experiment description, for example, by designing an experiment in TraceLab.

3 Background and Related Work

This section provides the background details about TraceLab as an environment for SM research and compares and contrasts TraceLab to other research tools specific to other domains.

[1] http://web.soccerlab.polymtl.ca/tefse09/Challenge.htm

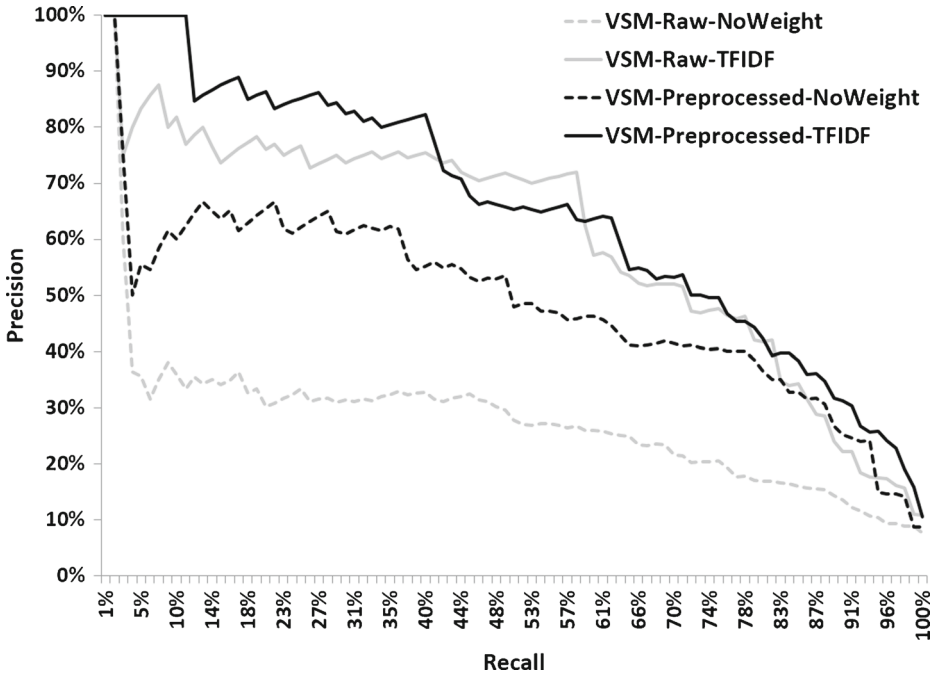

Fig. 1 Precision-Recall curves for EasyClinic for recovering traceability links between use cases and classes using a VSM-based traceability technique and different preprocessing techniques (*raw* – gray color, *preprocessed* – black color) and weighting schemes (*no weight* – dash line, *tf-idf* – solid line)

3.1 TraceLab

TraceLab (Cleland-Huang et al. 2011, 2012; Keenan et al. 2012) is a framework designed to support the reproducibility of experiments in software engineering and software maintenance. More specifically, it provides a visual workbench (see Fig. 2) that allows researchers to create, evaluate, compare, and most importantly share experiments in SM research. TraceLab was developed at DePaul University in collaboration with researchers at the College of William and Mary, Kent State University, and University of Kentucky, and it is already being used by numerous users throughout the world (see Fig. 5 for a geographical distribution of these users).

The heart of a TraceLab experiment lies in its workflow of *components* (see Fig. 2(1)). Components are reusable user-defined code units that are designed to accomplish a very specific task. They exchange data with other components through their inputs and outputs via shared memory. The components are represented in TraceLab as ovals (see examples from Figs. 2(1) and 3).

An *experiment* is a collection of components (or nodes) connected in the form of a precedence graph. The execution of an experiment begins at the "Start" node and continues along every path until the "End" node is reached, thus completing the experiment. Since it is a precedence graph, unless otherwise specified, each node must wait for all of the incoming edges to complete before executing. This ensures that the previous techniques have completed their execution and the correct data is available. The execution of components in TraceLab was designed to be parallelizable. Each component is given its own copy of the data and is run in a separate thread. The main reason for this design decision was to ensure that running

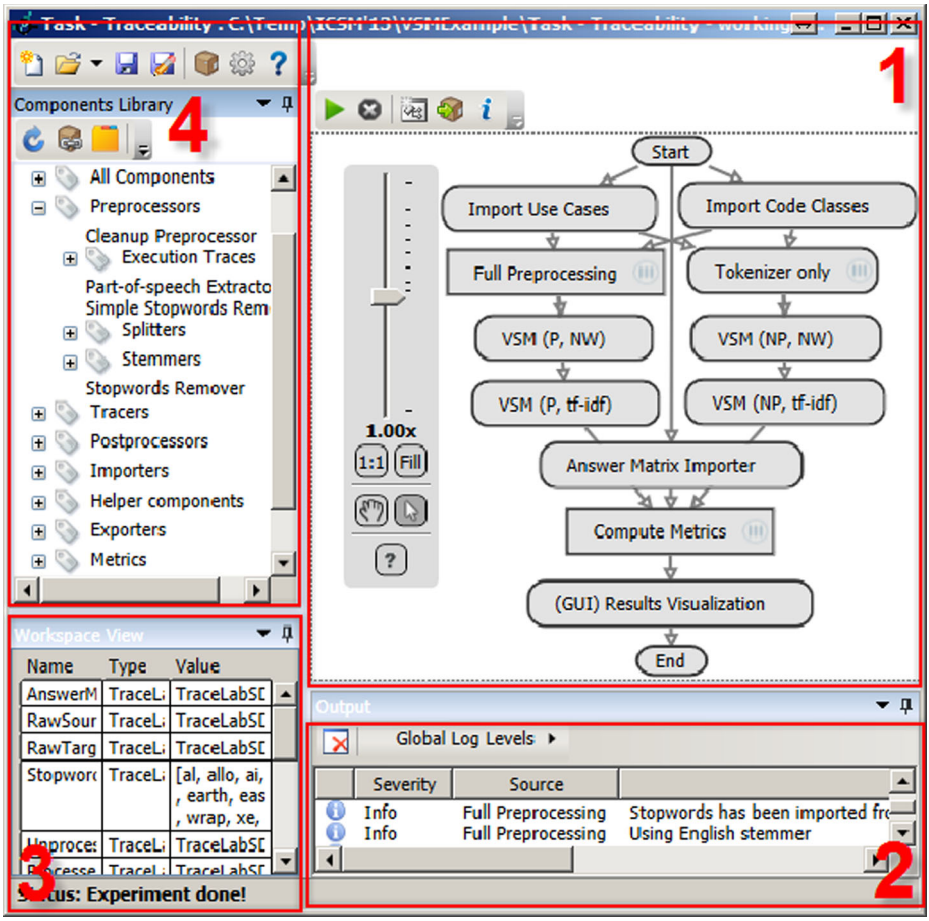

Fig. 2 The four "quadrants" of TraceLab in clockwise order from top-right are (*1*) the sample TraceLab experiment that implements our motivating example in Section 2; (*2*) an output window for reporting execution status of an experiment; (*3*) the Workspace containing the data and the values of the experiment; and (*4*) the *Component Library*

experiments will not encounter any errors caused by nondeterministic behavior triggered by race conditions, which means that developers do not have to worry about race conditions when designing their own custom components. Therefore, when two components branch out from a parent component (e.g., components "Import Use Cases" and "Import Code Classes" in Fig. 2(1)) they each will run concurrently and independently. This feature is built into the TraceLab framework, and it will be automatically applied to all execution paths of an experiment.

TraceLab was designed to run experiments fully automatically and without any interaction from the user. Moreover, the same experiment can be applied to multiple datasets, making it a great tool for batch experiment processing. A TraceLab experiment essentially takes as input (i) data (which, depending on the type of experiment was extracted from software systems, version control systems, issues tracking systems, execution traces, etc.) and (ii) an oracle (or ground truth) that was also generated using external tools or human support. Next, this data is

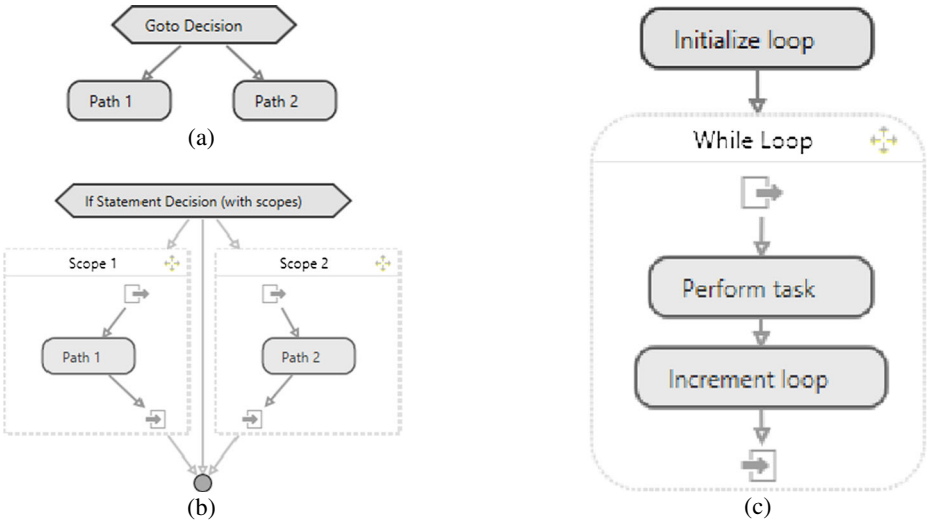

Fig. 3 Control flow options provided by TraceLab: **a** Goto decision, **b** If statement and (**c**) While loop

typically imported, converted to appropriate internal TraceLab datatypes, processed, and the evaluated based on a set of predefined metrics and the oracle. The results of an experiment could be visualized or exported for further analysis.

3.1.1 TraceLab Features

TraceLab provides many control elements that allow flexibility when designing an experiment. For example, *Goto decisions* (see Fig. 3(a)) allow execution redirection to any of the outgoing nodes based on a given condition. *If statement decisions* (see Fig. 3(b)) provide additional

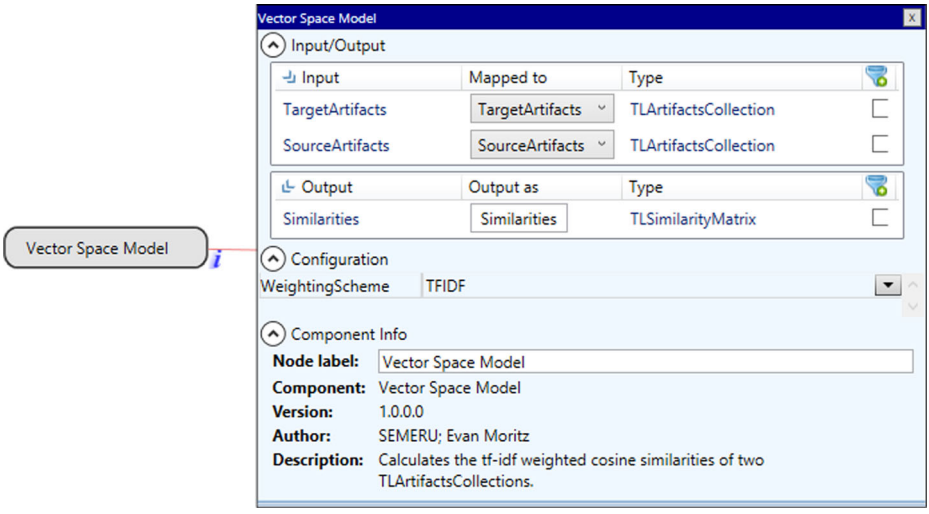

Fig. 4 Information pane for the Vector Space Model TraceLab component (left). The information pane shows the inputs and outputs, settings (e.g., weighting scheme) and other metadata information

control, by allowing execution of one of a number of sub-graphs (called scopes) based on a given condition. Scopes provide independent experiment sub-graphs that execute in their own namespace and once completed, provide control to the parent graph. Similarly, *While loops* (see Fig. 3(c)) repeatedly execute the scope as long as the given condition is true.

The *workspace* (see Fig. 2(3)) is the data-sharing interface that allows components to communicate with the preceding and following nodes. Components can load and store data to and from the workspace only for their declared inputs and outputs. Data may also be read from the workspace for use in a control-flow node. Any information in the workspace may be serialized to disk as an XML file for later use or debugging purposes. Additionally, any data type with a viewer (including all standard types) can be viewed from the workspace by clicking on their workspace entry. There are already a large number of predefined datatypes and components to handle these datatypes (e.g., importers, exporters, etc.), but if needed, researchers can adapt existing datatypes or create custom ones to fit their needs. Some of the predefined data types include TLArtifact (i.e., a generic data type that can represent any textual software artifact, such as requirements, design specifications, UML diagrams and defect logs, test cases, or software code; it has two fields, namely ID and textual information), TLArtifactsCollection (i.e., a collection of TLArtifacts), TLSimilarityMatrix (i.e., a datatype that represents the set of links from source artifacts to target artifacts with assign probability score of their relationship; this matrix can be used either as standard similarity matrix, an answer matrix, a traceability matrix, etc.). Moreover, the predefined data types include Lists, HashTables, Dictionaries, etc., which can be used for various tasks ranging from representing stop words to storing different values for the Box Plot points used to represent the results. In our experiments, we used a Program Dependence Graph (PDG), which stores basic information about the nodes and dependencies in the graph. However, since most applications that use a PDG require various types of information, practitioners are free (i) to use our PDG as is, (ii) to customize it for their unique needs (i.e., refining it), or (iii) to create a brand new PDG data type. In either case, an existing experiment that was exported and shared through TraceLab's packaging feature will still reference the correct component, regardless of the existence of other versions of the component. In other words, a shared experiment will still reference the "old" PDG (i.e., the one that was used for creating the experiment), and will not be affected by the newly created PDG (which can be part of future experiments).

The status of an executing experiment is reported in the *Output view* (see Fig. 2(2)) in the form of messages displayed to the user. The messages have different levels of severity, such as info, trace, debug, warning, and error. Each message displays the component name, severity, custom message provided by the author of the component (if any), and optionally an exception dialogue describing an uncaught exception and a stack trace.

A major contribution of this paper is a *Component Library*, designed to implement a wide range of SM techniques that can be easily accessed from TraceLab (see Fig. 2(4)) to build and execute experiments. The *Component Library* is included in the distribution of the official TraceLab release.

The component library provides a set of tools and techniques to researchers for using in experiments. Components may be categorized by multiple tags, both by component developers and users. In order to use a component in an experiment, the user only needs to drag-and-drop the component from the component library into the experiment (see Fig. 2(1)) and connect it with the other components. Each component has a set of metadata that uniquely identifies it within TraceLab. The primary identifier is the component's name, which appears in the component library and on the component node within the experiment. Components contain additional information such as a description, author, versioning information (see Fig. 4), inputs and outputs. For example, if a component takes in two sets of artifacts and produces a ranked

Fig. 5 Distribution of TraceLab users worldwide as of April 2013. *Red dots* indicate core developers. *Blue dots* indicate active contributors. *Green dots* indicate alpha and beta testers

list of similarities between the two, it must explicitly declare two TLArtifactsCollection objects as input and declare a TLSimilarityMatrix as output (see Fig. 4). This allows TraceLab to evaluate the experiment graph before running it, checking for valid inputs and control-flow errors. For example, if a component declares an input that is not an output of any preceding components, TraceLab will catch the error before the experiment starts. In addition to inputs and outputs, a component can allow the configuration of specific settings (e.g., weighting scheme for the Vector Space Model – see Fig. 4).

An important feature of TraceLab consists of generating *composite components*. More specifically, a group of individual components that are often used together as a group to accomplish a specific task can be combined to form *composite components*. This feature provides an additional level of abstraction for common functionality, it improves the reusability of components in the same experiment and across different experiments, and it improves the readability of the graph experiment. An example of such a composite component is the node with rectangular edges labeled *Queries preprocessing* in Fig. 7, which performs preprocessing on a corpus of documents by encapsulating the functionality of various components, such as identifier splitting, stemming, and stop-words removal.

In addition to all these features, arguably the most important and distinctive feature provided by TraceLab is the *packaging* feature for encapsulating and sharing experiments. In order to share a TraceLab experiment, all of the necessary information (i.e., data,

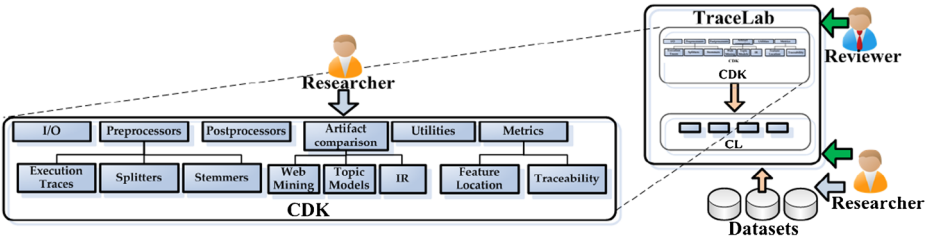

Fig. 6 Diagram of the hierarchy of the *CDK* in the context of TraceLab. *CDK* and *CL* are part of TraceLab. Researchers can contribute to the *CDK* and the datasets (*gray arrow*), and reviewers and researchers (*green arrow*) can use TraceLab to verify details of existing experiments

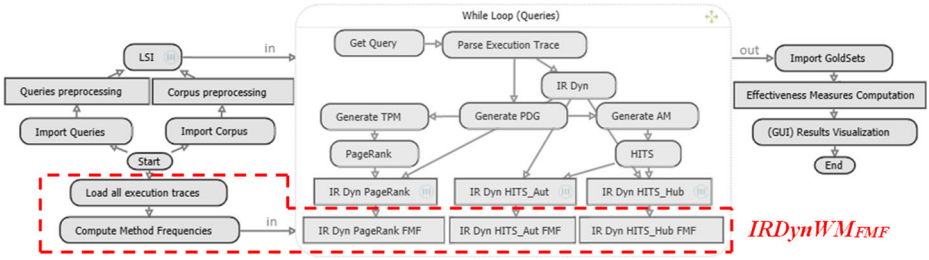

Fig. 7 TraceLab experiment for reproducing the *IRDynWM* FLT (Dit et al. 2013e) (without the components highlighted with *red dashed* border). TraceLab experiment for implementing $IRDynWM_{FMF}$ (all components, including the highlighted ones)

components and settings) must be included. Therefore, the packaging feature of TraceLab allows a user to encapsulate all the datasets and custom components used in the experiment, including all the dependencies and specific versions of datasets and components. The resulting self-contained experiment can be shared with the research community. The original experiment along with exact data and settings can be run by other researchers by unpacking and installing the original package experiment, using the associated functionality in TraceLab. Paths are relative to the package and they reference the data "inside" the package. Therefore, a shared package can be used as is by other researchers regardless of the location on their machine, because the paths are relative to the content of the package. The packaging feature not only allows to include data, but it can also reference existing packages which contain experiments and/or data. In other words, a researcher does not have to include the same data in different experiments, because she can choose to reference them. Once an experiment has been imported from a package, prior to running the experiment, one can change the configuration of the components that load/save data (i.e., if they want to use the same workflow on different data), or one can add/remove components to alter the workflow of the experiment if needed. In fact this one of the strengths of TraceLab. It ships with multiple importers, capable of importing data from multiple formats and converting it into standard data structures, while, at the same time, allowing the creation of new customized data importers. As a result, experiments can easily be run against different datasets. We would like to emphasize that there are two major reasons we include datasets in the packages. First, it ensures that the experiments are executable "out of the box", and we have observed that new users find it far easier to change a data source of an experiment that actually runs correctly, rather than having to find and configure the data to run the experiment. Second, one of the goals of TraceLab is to help new researchers get started, which implies providing them not only with the experiments, but also with the data.

It is important to highlight that researchers can create as many component types as they want (although we encourage reuse of components if possible). Each experiment consists of a particular dataset (which has a particular format and version), and a specific set of components (with their own version, configuration options) and dependencies between components. Once an experiment is created, and shared, it will reference specific versions of components and data types, even though some components may have evolved in the meanwhile. Therefore, the evolution of existing techniques will not affect the reproducibility of existing experiments.

We also note that our philosophy in creating TraceLab was to allow flexibility in the way components are designed, and we illustrated this with an example. We recently developed a new Weka component for a series of experiments, and this component is in effect a wrapper for the underlying Weka library. We needed to choose between (i) creating multiple TraceLab-

Weka components to perform specific classification functions (e.g., a J48 component, a Bayes Regression Classifier, etc.) or (ii) creating one generic component, which served as a general Weka wrapper. We opted for the second choice, which means that the component can be configured to run any classifier. This reduced the proliferation of components, but at the same time placed more onus on the component user to understand Weka's configuration parameters. The point is that TraceLab allows the experimenter to make such choices, and this was a deliberate design decision on our part.

3.1.2 Supported Languages

TraceLab ships with a software development kit (SDK) that allows users to define their own custom components and types in .NET languages, Java, and via plugins, R-Project (2013) Mathworks (2013) and Waikato (2013).

Any .NET language that compiles to a Dynamic Linked Library (DLL) may be used to create user-defined components and types. This includes Visual Basic, C++, C#, and F#.

Developers can create user-defined TraceLab components and types in Java using IKVM.NET.[2] After compiling the Java components, the JAR file is converted to a DLL through IKVM. Thus, when called in TraceLab, the Java code is actually run in the IKVM virtual machine.

In addition to .NET and Java, TraceLab components can execute R code. Although tools like R.NET[3] exist for running R code in .NET languages, they impose additional external dependencies on TraceLab and the development environment. Moreover, TraceLab has no built-in mechanism for recognizing components written in R. To overcome this issue, we have created a lightweight language plugin for R (named RPlugin) that allows R scripts to be run from TraceLab. The component classes are written normally in .NET, and any R scripts that need to be run interface with the plugin. RPlugin makes calls to an existing implementation of R and has a framework for passing data and running scripts in R. RPlugin is included with the TraceLab Component Library described section 5.

The developers of TraceLab have created a Matlab plugin similar to RPlugin that can run Matlab scripts from .NET.

3.2 Comparing TraceLab with Other Tools

There are many other frameworks and tools that have been designed to support research in other domains, such as information retrieval, machine learning, data mining, and natural language processing, among others. Consequently, reuse of third party tools or APIs is a common practice for constructing experiments and building research infrastructure in software evolution and maintenance. For example, a common scenario is to reuse WEKA for implementations of machine learning classifiers, R for statistical analysis, or MALLET for topic modeling. However, these tools/APIs were not built to support research on software evolution and maintenance. Moreover, most of the tools were conceived as extensible APIs and only few of them provide features such as experiment composition by using a data-flow GUI, new components implementation, or easy sharing/publishing of experiments; moreover, not all of them can be used across multiple platforms. Table 1 compares TraceLab to some similar tools that also use a data-flow oriented GUI.

[2] http://www.ikvm.net/
[3] https://rdotnet.codeplex.com/

Table 1 Comparison of TraceLab with other related tools (columns). The features (rows) are as follows: 1) data-flow oriented GUI [**Y**es / **N**o]; 2) Type of application [**D**esktop / **W**eb / **API**]; 3) License type [**C**ommercial / **O**pen source / **F**ree online access]; 4) Tool allows saving and loading experiments [**Y**es / **N**o]; 5) Tool allows creating composite components [**Y**es / **N**o / **P**rogrammatically]; 6) Tool has a component "market" where developers can contribute with their own components [**Y**es / **N**o]; 7) Programming language that can be used to build new components; 8) The platforms were the tool could be used [**S**oftware **A**s **A** **S**ervice, **W**indows, **L**inux, **M**ac]

Feature \ Tool	R Project	Matlab	Simulink	Weka/Rapid. Miner	Gate	Kepler	FETCH	Taverna	TraceLab
GUI	N	Y	Y	Y	N	Y	N	Y	Y
Type	API, D	API, D	D	API, D	API, D	API, D	D	D, W	API, D
License	O	C	C	O	O	O	O	O	O
Save/Load exp.	Y	Y	Y	Y	Y	Y	N	Y	Y
Composite component	P	P	Y	N	P	Y	N	Y	Y
Component Market	Y	Y	Y	N	Y	Y	N	Y	Y
Programming Language	R	Matlab Java C	C/C++ Matlab Fortran	Java	Java	R C Matlab Java	RML (Other)	Any	Java R .NET languages (e.g., C#) Matlab
Platforms	W, L, M	W, L, M	W, L, M	W, L, M	W, L, M	W, L, M	W, L	W, L, M	W, L, M

The R Project R-Project (2013) is a programming language and environment designed to perform statistical computing tasks on large-scale data. The tool is primarily command based, with the ability to produce charts and graphs. There are a multitude of user-contributed libraries for performing specialized tasks, including a variety of common software engineering research tasks. However, R does not feature an environment for designing experiments using a GUI-based workflow and can be difficult to reproduce when shared due to dependencies on a large number of libraries and different versions. Additionally, researchers must learn a new programming language when performing experiments in R.

Mathworks (2013) offers a programming language in an interactive environment geared towards numerical computation, data analysis, visualization (e.g., 2D and 3D visualization) and programming. Matlab was designed to be used in diverse areas, ranging from signal processing, image processing, testing and measuring, computational finance and many others.

Mathworks (2013) is a Matlab-based tool for simulation and model-based design of embedded systems. In Simulink, a model is composed of subsystems (i.e., a group of blocks) or individual blocks, and the blocks can be implemented using Matlab, C/C++, or Fortran. Building the model can be accomplished using drag-and-drop of blocks and making connections between them, which is similar to the way TraceLab allows to build experiments.

Waikato (2013) is a collection of machine learning algorithms that are packaged as an open source Java library that also allows running the algorithms using a graphical user interface (GUI). One of the WEKA modules is the *KnowledgeFlow*, which provides the user with a data-flow oriented GUI for designing experiments. As in TraceLab, the components in the KnowledgeFlow are categorized by tasks (DataSources, DataSinks, Filters, Classifiers, Clusterers, Associations, Evaluation, Visualization), and there is a layout canvas for designing experiments by dragging, dropping, and connecting components. New components can be added to WEKA by extending or modifying the library using Java, and the experiments can be saved and loaded for being executed in the WEKA Experimenter module.

RapidMiner Rapid-I (2013) is a data mining application that provides an improved GUI for designing and running experiments. It includes a reusable library for designing experiments and running them and it fully integrates WEKA as the machine learning library.

GATE (Sheffield 2011) provides an environment for text processing that includes an IDE with components for language processing, a web application for collaborative annotation of document collections, a Java library, and a cloud solution for large scale text processing.

Kepler (2013) is a tool that follows the same philosophy as TraceLab. By using Kepler, it is possible to build, save, and publish experiments/components using a data-flow oriented GUI. It is also possible to extend Kepler because of its collaborative-project nature. However, the main difference with TraceLab is that Kepler was conceived as a tool for experiments in sciences such as Math or Physics.

FETCH (2014) is a set of third-party open source tools linked in a pipeline to support program analysis of large C/C++/Java software systems. Fetch does not allow researchers to design experiments or extend components. Instead, it is a command-line based tool that applies several analyses to a software system and generate reports (i.e., charts, tables, files) describing the results of the analysis.

Taverna (2014) is a workflow-based tool for designing experiments, by connecting components deployed as web services. The components are imported into Taverna through the web service's WSDL (Web Service Description Language). Therefore, Taverna is independent of the programming language, and researchers have to write their components on any language and publish them as a discoverable web service. However, Taverna does not provide an IDE for implementing/publishing web services. Taverna has a workbench for designing the workflows, a server for the remote execution of workflows when required, and a command line tool for workflow execution from a terminal.

Although TraceLab is not specialized on simulation, natural language processing, or machine learning, it was specifically designed to allow software engineering and maintenance researchers the possibility to (i) *develop* and *share* their own components/experiments, and (ii) to ensure the *reproducibility* of their results. During the design of TraceLab, a traditional desktop application was preferred over a Service-Oriented Architecture solution because we believed that the overhead of services (i) may not suit the kinds of experiments that might be conducted in SE, and (ii) could have introduced an additional burden for the user to create and maintain services. Furthermore, service composition adds additional overheads, which are not suited to some experiments in the traceability domain as well as other SE domains. Hence, our goal was to create a local solution (to avoid confidentiality issues) to allow users to quickly and easily create and compose components. Although TraceLab was initially implemented in C# and supported only Windows, the latest version of TraceLab is cross-platform and supports all major OS platforms (e.g., Window, MacOS and Linux). However, we made publicly available on GitHub the Windows version of TraceLab, and we will open-source the cross-platform version of TraceLab after it goes through the incubation stage. In order to achieve this support for all OS platforms, TraceLab was compiled using Mono,[4] the open source, cross platform .NET framework.

To implement the TraceLab components, researchers can use Java, any .NET language (e.g., C#, VB, C++), R or Matlab. For the .NET languages, either Microsoft Visual Studio or Mono can be used.

4 Mapping Study of Software Maintenance Techniques

In this section we present the methodology, analysis, and results of a mapping study (Kitchenham et al. 2011) aimed at identifying a set of techniques from particular areas of SM, which could be implemented as TraceLab experiments in order to constitute an initial practical body of knowledge that would benefit the SM research community. Moreover, these identified techniques were reverse engineered into basic modules that we implemented as TraceLab components, in order to generate a *Component Development Kit* (see Section 5.1) and a *Component Library* (see Section 5.2) that serves as a starting point for any interested researcher to implement new techniques or build upon existing ones.

For our study, we used the systematic mapping process described by Petersen et al. (2008). The process consists of five stages: 1) defining the research questions of the study, 2) searching for papers in different venues, 3) screening the papers based on inclusion and exclusion criteria in order to find relevant ones, 4) classifying the papers, and 5) extracting data and then generating the systematic map.

A mapping study is different from a systematic literature review in that literature reviews aim to answer a specific research question by extracting and analyzing the results of primary studies (Kitchenham et al. 2011), for example, a review of studies analyzing development effort estimation techniques to see which ones work the best (Jørgensen 2004). In contrast, mapping studies attempt to address more abstract research topics by classifying the methodologies and findings into general categories. Mapping studies are useful to the research community in that they provide an overview of trends within the search space (Petersen et al. 2008). Furthermore, they may be used as a starting point by researchers looking to improve the field by describing common methodologies and perhaps discovering untapped areas that others have missed.

[4] www.mono-project.com/

4.1 Defining the Research Question

Our goal is to identify a set of representative techniques from specific areas of SM, and then use them to generate TraceLab components and experiments to accelerate and support research in SM. Thus, we defined the following research questions (RQs) for the mapping study:

> *RQ_1: What types of techniques are common to experiments in software evolution and maintenance research?*
> *RQ_2. What individual techniques are used across many SM experiments?*
> *RQ_3. How do experiments in SM research differ across different sub-domains?*

RQ_1 attempts to identify high-level categories containing groups of techniques designed to perform similar research tasks. RQ_2 focuses on individual techniques and aims to identify the most common techniques used in experiments in the mapping study. RQ_3 is intended to compare and contrast how techniques are used in different high-level research tasks, such as traceability link recovery or feature location.

These three RQs can be reformulated into a single main research question as follows:

> *RQ: Which SM techniques are suitable to form an initial actionable body of knowledge that other researchers could benefit from?*

In particular, we focused on a subset of SM areas in which the authors have expertise. This allowed us to generate an initial and extensible body of knowledge that could support the research community. The SM research community, can contribute to the body of knowledge by continually adding new techniques and components.

4.2 Conducting the Search

In order to find these techniques, we narrowed the search space to the publications from the last 10 years of a subset of journals and software engineering conferences. In addition, in our search we incorporated the "snowballing" discovery technique (i.e., following references in the related work) discussed by Kitchenham et al. (2011). The list of journals and conferences from which we selected at least one paper in our mapping study is presented in Table 2. Additional information can be found in our online appendix.

4.3 Screening Criteria

The primary *inclusion* criterion consisted of identifying whether the research paper described a technique that addressed one of the following maintenance tasks: traceability link recovery, feature location, program comprehension and duplicate bug report identification. In most cases, this information was determined by the authors of this paper by reading the title, abstract, keywords, and if necessary the introduction and the conclusion of the investigated paper.

The *exclusion* criteria were as follows. First, we discarded techniques that could not have been implemented effectively in TraceLab due to various reasons, such as (i) lack of sufficient implementation details, (ii) lack of tool availability or (iii) the technique was not fully automated, and would require interaction with the user. Second, we did not implement complex techniques that would have required a lengthy development time, or techniques that are outside the expertise of the authors. Third, we discarded techniques with numerous

Table 2 List of Journals and Conferences for which we identified at least one paper in our mapping study

Abbreviation	Venue name	# papers
ASE	Automated Software Engineering	1
CSMR	European Conference on Software Maintenance and Reengineering	2
EMSE	Empirical Software Engineering	1
ICSE	International Conference on Software Engineering	6
ICPC	International Conference on Program Comprehension	9
ICSM	International Conference on Software Maintenance	3
MSR	Working Conference on Mining Software Repositories	2
TEFSE	International Workshop on Traceability in Emerging Forms of Software Engineering	1
TSE	Transactions in Software Engineering	1
WCRE	Working Conference on Reverse Engineering	1
	Total	27

dependencies to deprecated libraries or other techniques, as our goal was to implement the most popular techniques that can be incorporated or built upon.

4.4 Classification

In our mapping study we used two independent levels of classification. The first one consisted of categorizing the papers based on the SM task (e.g., traceability link recovery, feature location, program comprehension and detecting duplicate bug reports) they presented (see Section 4.3). This classification was targeted at answering RQ_3.

The second level of classification was identifying common functionality between the basic building blocks used in an approach (e.g., all the functionality related to identifier splitting, stemming, stopword removal and others, were grouped under "preprocessing"). This level of classification is necessary for answering RQ_1 and RQ_2.

4.5 Data Extraction

The list of papers that we identified in our study is presented in Table 3 in the first column along with the Google Scholar citation count as of December 1, 2013 (second column). The papers are grouped by the primary SM tasks they address, and are sorted chronologically.

The remaining columns constitute the individual building blocks and components we identified in each approach, grouped by their common functionality. A checkmark (✓) denotes that we implemented the component in the *CL*. An *X* denotes that the code related to the components appears in the approach, but is not implemented in the *CL* at this time (see Section 5.2 and Section 8).

Table 3 shows only a subset of the information. For the complete information, we refer the interested reader to our online appendix.

5 Component Library and Development Kit

From the 27 papers identified in the mapping study, we reverse engineered their techniques in order to create a comprehensive library of components and techniques with the aim of

Table 3 Mapping study results (first column) and implementation of these techniques in the *CDK* (✓means that the component from the first row is implemented in *CDK* and *X* means is not yet implemented in *CDK*)

Technique Year / Venue / Name / Ref	Google scholar citation count	Preprocessing									
		Bag-of-words tokenizer	Stopwords remover	Porter stemmer	CamelCase splitter	Execution trace logger	Dependency graph generator	Samurai splitter	Smoothing filter	Snowball stemmer	Part-of-speech tagger
Traceability link recovery											
2008.ICPC.Abadi (Abadi et al. 2008)	58	✓	✓	✓	.	.	.	.	.	.	.
2009.ICPC.Capobianco (Capobianco et al. 2009)	26	✓	✓	✓	✓	.	.	.	.	.	✓
2010.ICPC.Oliveto (Oliveto et al. 2010)	71	✓	✓	✓	.	.	.	.	.	.	.
2010.ICSE.Asuncion (Asuncion et al. 2010)	87	✓	✓	✓	.	.	.	.	.	.	.
2011.ICPC.DeLucia (De Lucia et al. 2011)	15	✓	✓	✓	✓	.	.	.	✓	.	.
2011.ICSE.Chen (Chen et al. 2011)	5	✓	.	.	.	.	.	.	.	.	.
2011.ICSM.Gethers (Gethers et al. 2011)	30	✓	✓	✓	✓	.	.	.	.	.	.
2013.CSMR.Panichella (Panichella et al. 2013b)	5	✓	✓	.	.	.	✓	.	.	.	.
2013.ICSE.Panichella (Panichella et al. 2013a)	10	✓	.	.	.	.	.	.	.	.	.
2013.TEFSE.Dit (Dit et al. 2013c)	3	✓	.	.	.	.	.	.	.	.	.
Feature location											
2004.WCRE.Marcus (Marcus et al. 2004)	277	✓	.	.	✓	.	.	.	.	.	.
2007.ASE.Liu (Liu et al. 2007)	104	✓	✓	.	✓	X	.	.	.	.	.

Table 3 (continued)

Technique Year / Venue / Name / Ref	Google scholar citation count	Preprocessing									
		Bag-of-words tokenizer	Stopwords remover	Porter stemmer	CamelCase splitter	Execution trace logger	Dependency graph generator	Samurai splitter	Smoothing filter	Snowball stemmer	Part-of-speech tagger
2007.TSE.Poshyvanyk (Poshyvanyk et al. 2007)	212	✓	✓	.	✓	.	.	.	.	.	.
2009.ICPC.Revelle (Revelle and Poshyvanyk 2009)	37	✓	.	.	.	X	✓	.	.	.	.
2009.ICSM.Gay (Gay et al. 2009)	48	✓	✓	✓	✓	.	.	.	.	.	.
2011.ICPC.Dit (Dit et al. 2011)	29	✓	✓	✓	✓	X	.	X	.	.	.
2011.ICPC.Scanniello (Scanniello and Marcus 2011)	13	✓	✓	✓	✓	.	✓	.	.	.	.
2011.ICSM.Wiese (Wiese et al. 2011)	6	✓	.	✓	.	.	.	.	.	✓	.
2012.ICPC.Dit (Dit et al. 2012)	11	✓	✓	✓	✓	X	.	.	.	.	.
2013.EMSE.Dit (Dit et al. 2013e)	9	✓	✓	✓	✓	X	✓	.	.	.	.
Program comprehension											
2009.MSR.Enslen (Enslen et al. 2009)	65	✓	.	.	✓	.	.	X	.	.	.
2009.MSR.Tian (Tian et al. 2009)	38	✓	✓	.	✓	.	.	.	.	.	.
2010.ICSE.Haiduc (Haiduc et al. 2010)	32	✓	✓	✓	✓	.	.	.	.	.	.
2012.ICPC.DeLucia (De Lucia et al. 2012)	7	✓	✓	✓	✓	.	.	.	.	.	.

Table 3 (continued)

Technique Year / Venue / Name / Ref	Google scholar citation count	Preprocessing									
		Bag-of-words tokenizer	Stopwords remover	Porter stemmer	CamelCase splitter	Execution trace logger	Dependency graph generator	Samurai splitter	Smoothing filter	Snowball stemmer	Part-of-speech tagger
Identify duplicate bug rep.											
2007.ICSE.Runeson (Runeson et al. 2007)	185	✓	✓	✓	.	.	.	.	.	.	.
2008.ICSE.Wang (Wang et al. 2008)	188	✓	✓	✓	.	X	.	.	.	.	.
2012.CSMR.Kaushik (Kaushik and Tahvildari 2012)	9	✓	✓	✓	.	.	.	.	.	.	.
Total:	1,580	27	20	17	14	6	4	2	1	1	1

Artifact comparison							Metrics			Postprocessing				Other
Latent Semantic Indexing	Vector Space model	Latent dirichlet allocation	Jensen-Shannon divergence	Relational topic model	HITS	PageRank	Precision / Recall metrics	Effectiveness measure	Principal component analysis	Execution trace extractor	Affine transformation	O-CSTI	UD-CSTI	Genetic algorithm
Traceability link recovery														
✓	✓	.	✓	.	.	.	✓	.	.	.	.	.	.	.
✓	.	.	✓	.	.	.	✓	.	.	.	.	.	.	.
✓	✓	✓	✓	.	.	.	✓	.	✓	.	.	.	.	.
✓	.	✓	.	.	.	.	✓	.	.	.	.	.	.	.
✓	✓	.	.	.	.	.	✓	.	.	.	.	.	.	.
.	✓	.	.	.	.	.	✓	.	.	.	.	.	.	.
.	✓	.	✓	✓	.	.	✓	.	✓	.	✓	.	.	.
.	✓	.	✓	.	.	.	✓	.	.	.	.	✓	✓	.
.	.	✓	.	.	.	.	✓	✓	.	.	.	.	.	✓
.	.	✓	.	.	.	.	✓	.	.	.	.	.	.	✓

Table 3 (continued)

Artifact comparison							Metrics			Postprocessing				Other
Latent Semantic Indexing	Vector Space model	Latent dirichlet allocation	Jensen-Shannon divergence	Relational topic model	HITS	PageRank	Precision / Recall metrics	Effectiveness measure	Principal component analysis	Execution trace extractor	Affine transformation	O-CSTI	UD-CSTI	Genetic algorithm
Feature location														
✓	.	.	.	.	.	.	✓	.	.	.	.	.	.	.
✓	.	.	.	.	.	.	.	✓	.	✓	✓	.	.	.
✓	.	.	.	.	.	.	.	✓	.	.	✓	.	.	.
✓	.	.	.	.	.	.	.	.	.	✓	.	.	.	.
.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.
✓	.	.	.	.	.	.	.	✓	.	✓	.	.	.	.
.	✓	.	.	.	.	.	.	✓	.	.	.	.	.	.
.	✓	.	.	.	.	.	.	.	.	.	.	.	.	.
✓	✓	.	.	.	.	.	.	✓	.	✓	.	.	.	.
✓	.	.	.	.	✓	✓	.	✓	.	✓	.	.	.	.
Program comprehension														
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	✓	.	.	.	.	✓	.	.	.	.	.	.	.
✓	.	.	.	.	.	.	.	.	.	.	.	.	.	.
✓	✓	✓	.	.	.	.	.	.	.	.	.	.	.	.
Identify duplicate bug rep.														
.	✓	.	.	.	.	.	✓	.	.	.	.	.	.	.
.	✓	.	.	.	.	.	✓	.	.	.	.	.	.	.
✓	✓	✓	.	.	.	.	✓	.	.	.	.	.	.	.
14	14	7	5	1	1	1	14	7	2	5	3	1	1	2

providing the necessary functionality that SM researchers would need to reproduce experiments and create new techniques.

This process resulted in generating (i) a ***C**omponent **D**evelopment **K**it* (*CDK*) that contains the implementation of all the SM techniques from the study, (ii) a ***C**omponent **L**ibrary* (*CL*) that adapts the *CDK* components to be used in TraceLab and (iii) the associated documentation and usage examples for each.

5.1 Component Development Kit

The *Component Development Kit* (*CDK*) is a multi-tiered library of common tools and techniques used in SM research. These tools are organized in a well-defined hierarchical structure and exposed through a public API. The intent of this compilation is to aid researchers in reproducing existing approaches and creating new techniques for *software maintenance and evolution*. Therefore, the appropriate name for this CDK would be *Component Development Kit-Software Maintenance and Evolution (CDK-SME)*, in order to distinguish it from other CDKs (e.g., related to *requirements engineering*) that will be developed in the near future by other groups in the research community surrounding TraceLab (see Fig. 5). However, for brevity, and because in this paper we are only discussing the CDK for software maintenance and evolution, throughout the paper we will refer to *CDK-SME* as *CDK*.

By providing access to tools and techniques related to software maintenance and evolution tasks, the *Component Development Kit* facilitates the research evaluation process, and researchers no longer have to start from scratch or spend time adapting their pre-existing tools to a new project. Furthermore, researchers can use combinations of these tools to create new techniques and drive new research.

At the top level, the *CDK* is separated into categories of high-level tasks, such as I/O, preprocessing techniques, artifact comparison techniques, and metrics calculations (see Fig. 6). Those categories are then further broken down as needed into more specific tasks. This design aids technique developers in locating relevant functionality quickly and easily, as well as providing base points for integrating new functionality in the future. The high-level categories are described as follows:

Data preprocessing techniques primarily convert the raw data into a different form that will be used in other steps of the approach. For text-based approaches, preprocessing typically involves extracting comments and identifiers from source code, removing stopwords, stemming, as well as other methods for text manipulation. For structural approaches, preprocessing could involve parsing an execution trace or calculating a static dependency graph. The preprocessing techniques are usually the first to run, before any other steps of the approach.

Artifacts comparison techniques encapsulate all the techniques that implement any kind of comparison between software artifacts to determine relationships between them. These techniques usually take in a set of software artifacts (such as source code or requirements documents) as input and produce a set of suggested relationships between documents. These suggestions may include a confidence score, textual similarity scores, etc., which are useful for ranking the set of input artifacts based on various criteria.

The *metrics* category encapsulates the measures by which an approach is evaluated. These are used to determine the accuracy (or performance) of a technique and to compare it with other techniques. Examples of such metrics include precision, recall, effectiveness, etc.

Guided by findings from the mapping study, we evaluated each technique based on coverage, usefulness, and perceived difficulty and effort in implementation. In addition to our design goals of providing a clean and easy to use API, another goal was to minimize the

number of external dependencies necessary to implement the technique. As such, some techniques that have numerous external dependencies were left out.

5.2 Component Library

The *Component Library* (*CL*) is comprised of metadata and wrapper classes registering certain functionality as components in TraceLab. It acts as a layer in between TraceLab and the *CDK*, adapting the functionality of the *CDK* to be used within TraceLab. A typical component will import data from TraceLab's data sharing interface (the *Workspace*), call various functions on the data using the *CDK*, and then store the results back to the *Workspace*.

To register a component in TraceLab, a class must inherit from the BaseComponent abstract class defined in the TraceLab SDK. All components must override the Compute() method which contains the desired functionality of the component within the context of a TraceLab experiment. Component classes may also override PreCompute() and PostCompute() to pre-allocate and dispose of resources (these methods are called immediately before and after the Compute() method).

Furthermore, all components have a component declaration attribute (or annotation in Java terminology) that describes information about the component. For example, the [Component] attribute specifies information about the component's name, description, author, version, and optional configuration object. The configuration object is responsible for all the settings associated with the component (e.g., weighting schema for Vector Space Model, input path for a component loading a corpus, etc.). Any inputs and outputs from and to the workspace must be declared with individual [IOSpec] attributes describing the input or output name and data type. Lastly, components may optionally declare [Tag] attributes for automatic categorization in the component library. The declaration of all these attributes serve two purposes, namely to (i) allow the class to be registered in TraceLab as a component, and (ii) to ensure that a component can only be connected with a compatible component. Figure 4 shows an example of a component that used three [IOSpec] attributes (two for input and one for output) to define the name (e.g., *TargetArtifacts*, *SourceArtifacts* and *Similarities*) and data type (e.g., TLArtifactsCollection and TLSimilarityMatrix) of inputs and outputs. In addition, the metadata specific to the component, which was defined using the [Component] attribute is presented in the lower right corner of the figure.

After compiling, libraries containing components should be placed in a registered component directory, in order to allow TraceLab to recognize them and make them available in the Component Library (see Fig. 2(1)). These directories are defined in TraceLab's settings menu and user-defined directories can be added or removed as needed.

In addition to custom components, the TraceLab SDK allows users to define custom data types. These user-defined data types must declare a [WorkspaceType] attribute in order for TraceLab to recognize them as workspace types that can be used in the workspace. These types must also declare a [Serializable] attribute to allow data to be transferred between the workspace, components, and disk. It is important to note that any custom data types that do not need to be used in the workspace (e.g., intermediate data used within a component) do not need to be registered with TraceLab. Libraries containing types must also be placed in a registered directory containing types, which is usually separate from the components library. Workspace types may also provide custom visualizations for inspecting the data after an experiment has run.

The *Component Library* uses the same structure as the *CDK* (see Fig. 6), providing a mapping from TraceLab to the *CDK*. Components can be organized in TraceLab through the use of hierarchically organized developer and user *Tags*, another feature of the TraceLab SDK. Components are grouped via Tags into the same high-level tasks as the *CDK*.

From the building blocks of the *CDK* identified in the mapping study, we implemented 25 out of 51 as TraceLab components. In many cases, this was done as a one-to-one mapping from the *CDK* to the *CL*. However, some techniques could be broken down into more general ones, which were desirable for component re-use. For example, the Vector Space Model (VSM) is a straightforward technique, but there can be many variations on its implementation (see Section 2). We implemented a few weighting schemes (e.g., binary term frequency, tf-idf, and log-idf) and similarity functions (e.g., cosine, Jaccard), so that a component developer could pick and choose from the desired schemes.

Another example is the precision and recall metrics in traceability link recovery. Although this component consists of only one column in the mapping study, the *CDK* covers many of the commonly used metrics in the literature (e.g., precision, recall, average precision, mean average precision, F-measure, and precision-recall curves). Component developers could choose from any of these measures in their experiments.

5.3 Documentation

Documentation of the *CDK* and *CL* plays a key role in assisting researchers and component developers new to TraceLab. In addition to code examples and API references, documentation provides vital information about a program's functionality, design, and intended use. This adds a wealth of knowledge to someone who wants to use TraceLab and start designing new experiments from components. We provide this information in a wiki format on our website[5,6] which includes a developer guide, the CDK API reference, release notes, and code examples.

5.4 Extending the CDK and CL

The *CL* and *CDK* themselves are not the definitive collection of all the SM tools that researchers will ever need. However, their design and implementation in conjunction with TraceLab's framework provide a foundation for extending SM research in the future.

Both the *CL* and *CDK* are released under an Open Source license (GPL) in order to facilitate collaboration and community contribution. As new techniques are invented, they can be added to the existing hierarchy and thus into TraceLab.

In creating the *CL* and *CDK*, we leveraged TraceLab's ability to modify existing components or to create custom (i.e., user made) components that will fit the need of a researcher, through the TraceLab SDK. Researchers can also create adapt or modify existing datatypes or create new ones if needed. It is important to know that multiple versions of the same datatypes can exist, but once a particular version of a datatype or component is referenced in a particular experiment, that version of the datatype or component will be exported and shared with the community (using TraceLab's packaging feature – see Section 3.1.1), to ensure the reproducibility of an experiment even in the case of having multiple versions of the same datatype or component. As the body of SM techniques grows, researchers can utilize our components and extend them to new ones via the same process. Part of our future work will be dedicated to ensuring that existing components would be easier to discover and understand by other researchers (e.g., through proper documentation), to encourage reuse and reduce the number of overlapping components. Moreover, we will focus on establishing a process of incorporating user-made components into the *CL* and *CDK*, by establishing a standard of quality that each proposed component must satisfy.

[5] http://coest.org/coest-projects/projects/semeru/wiki
[6] https://github.com/CoEST

6 Reproducing Existing Experiments and Evaluating New Ideas Using the Component Library

This section presents the details of reproducing an existing feature location technique (FLT) (Dit et al. 2013e) using the *CDK* and the *CL* proposed in this paper. We describe the original technique, the details of reproducing it in TraceLab, and compare the results of the original and reproduced technique. We then illustrate the process of experimenting with two new ideas that are based on the reproduced technique. Moreover, we describe the details of reproducing two traceability link recovery experiments by Oliveto et al. (2010) and Gethers et al. (2011) using the proposed *CDK* and the *CL*.

6.1 Reproducing a Feature Location Technique

The FLT introduced by Dit et al. (2013e), called $IR_{LSI}Dyn_{bin}WM$ (or *IRDynWM* for short), was reproduced in TraceLab using a subset of components from the proposed *CL*. Please refer to Table 4 for a list of acronyms that are introduced throughout the remaining of this section. The high-level idea behind *IRDynWM* is to (i) identify a subset of methods from an execution trace with high or low rankings using advanced web mining analysis algorithms and to (ii) remove those methods from the results produced by the SITIR approach (Liu et al. 2007). The SITIR

Table 4 List of acronyms and their explanation for the feature location technique experiment used in Section 6

Feature location technique	Explanation
$IR_{LSI}Dyn_{bin}WM$	FLT introduced by Dit et al. (Dit et al. 2013e)
IRDynWM	Abbreviation for $IR_{LSI}Dyn_{bin}WM$
WM	WM stands for "Web Mining" and can be either *PageRank*, *HITS_Aut* or *HITS_Hub*
IRDyn	The FLT introduced by Liu et al. (Liu et al. 2007) that uses information retrieval (*IR*) and dynamic information (Dyn) from execution traces. This technique is also known as the SITIR approach
IRDynPageRank	An instance of the *IRDynWM* FLT that combines *IRDyn* with PageRank scores
IRDynHITS_Aut	An instance of the *IRDynWM* FLT that combines *IRDyn* with Authority scores from the HITS algorithm
IRDynHITS_Hub	An instance of the *IRDynWM* FLT that combines *IRDyn* with Hubs scores from the HITS algorithm
$IR_{LSI}Dyn_{bin}$	T_1 from (Dit et al. 2013e)
$IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{t80}$	T_2 from (Dit et al. 2013e)
$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{b60}$	T_7 from (Dit et al. 2013e)
$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{b80}$	T_{13} from (Dit et al. 2013e)
$IRDynWM_{FMF}$	Variation of *IRDynWM* which filters frequent methods that appear in execution traces (i.e., FMF stands for "Frequent Method Filter") and was introduced in Section 6.2.1
$IRDynWM_{NSF}$	Variation of *IRDynWM* which filters methods that do not appear in execution traces that exercise no scenario (i.e., NSF stands for "No Scenario Filter") and was introduced in Section 6.2.2

approach (or *IRDyn*) uses information retrieval (*IR*) techniques to rank all the methods from an execution trace (*Dyn*) based on their textual similarities to a maintenance task used as a query.

The *IRDynWM* FLT takes as input a description of a maintenance task in natural language (e.g., bug report description), the source code of the system, and an execution trace of a scenario that exercises the feature described in the maintenance task. The execution trace contains a list of methods that were executed during the scenario exercised by the user. The execution trace is processed and converted into a program dependence graph (PDG), where a pair of connected nodes represents a caller-callee relation between two methods from the execution trace. The PDG is used as an input for two link analysis algorithms, namely PageRank (Brin and Page 1998) and HITS (Kleinberg 1999), which generate a score for each node from the PDG (i.e., each method from the execution trace). PageRank produces one score for each method, which represents the popularity or importance of that method within the graph (Brin and Page 1998). HITS produces two scores for each method: (i) an authority score, based on the content of the method and the number of methods pointing to it (i.e., methods that are called by other methods should have a higher authority score), and (ii) a hub score, based on the outgoing links of a method (i.e., methods that call numerous other methods have higher hub values) (Kleinberg 1999). The different scores produced by PageRank and HITS are used to rank methods and identify the ones with high or low importance scores in order to remove them from the list of results produced by the SITIR (*IRDyn*) approach (Liu et al. 2007).

The reproduced *IRDynWM* FLT, where *WM = {PageRank* or *HITS_Aut* or *HITS_Hub},* is presented as a TraceLab experiment in Fig. 7 (see components in the upper part of the figure, which are not highlighted by the red dashed line).

The experiment uses TraceLab's *loop structure* (see the node labeled "While Loop (Queries)" in Fig. 7 which has two arrows labeled "in" pointing to it, and one arrow labeled "out" exiting it) to iterate through all the queries in the dataset and (i) retrieves and parses its execution trace (*Parse Execution Trace*), (ii) generates a program dependence graph based on the caller-callee relations identified in the trace (*Generate PDG*), (iii) generates a transition probability matrix for PageRank (*Generate TPM*) and applies PageRank (*PageRank*) to generate the importance scores, and similarly, it generates an adjacency matrix (*Generate AM*) used by HITS (*HITS*) to generate the authorities and hubs scores associated with these methods. The parsed methods from the execution trace are used by the *IRDyn* component to produce the results of the SITIR approach, and these results, along with the results produced by the *PageRank* component, are used to generate the results for the *IRDynPageRank* FLT (see *IR Dyn PageRank* component). Similarly, using the HITS authorities and hubs scores, the *IRDynHITS_Aut* and *IRDynHITS_Hub* FLTs are computed. Moreover, the components associated with the *IRDynWM* FLT can be configured with user defined thresholds for the percentage of methods to filter (Dit et al. 2013e).

The results produced by the replicated technique are the same as the ones reported in the original paper, even though the original technique used different implementations of Latent Semantic Indexing (LSI) (Deerwester et al. 1990), PageRank and HITS algorithms, as well as other scripts to compute the results. Figure 8 shows a subset of the results produced by our TraceLab implementation, which are the same as the ones that were reported in (Dit et al. 2013e) Fig. 4(c) for the jEdit dataset (see Fig. 10, which is a reproduction of (Dit et al. 2013e) Fig. 4(c)). Figure 8 represents the box plots of the *effectiveness measure* (i.e., the rank of the first relevant method in the list of results (Liu et al. 2007)) for the techniques T_1, T_2, T_7 and T_{13} corresponding to $IR_{LSI}Dyn_{bin}$, $IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{t80}$, $IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{b60}$, $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{b80}$ from (Dit et al. 2013e) Fig. 4(c), using the notation from (Dit et al. 2013e). For simplification, T_2, T_7, and T_{13} correspond to the *IRDynPageRank, IRDynHITS_Aut* and *IRDynHITS_Hub* respectively. These techniques were chosen as an

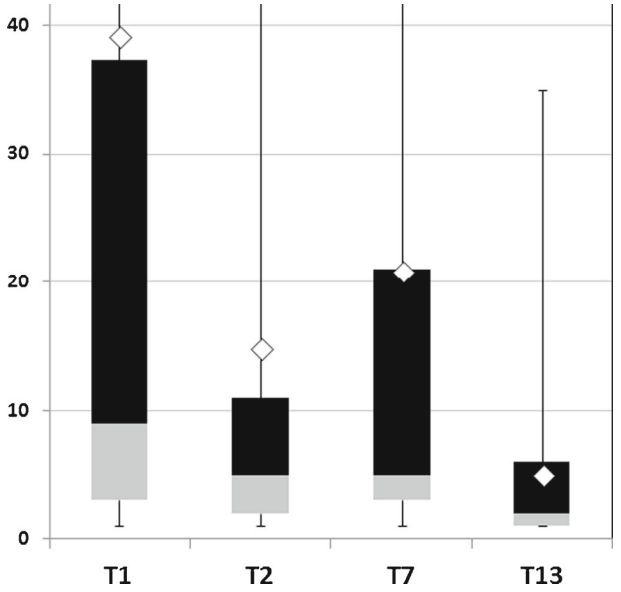

Fig. 8 *Box* plots of effectiveness measure (see y-axis) obtained from reproducing the experiments in (Dit et al. 2013e). The results of techniques T_1, T_2, T_7 and T_{13} correspond to $IR_{LSI}Dyn_{bin}$, $IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{t80}$, $IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{b60}$, $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{b80}$ from (Dit et al. 2013e) Fig. 4(c)

example in Fig. 8 because they produced the best results for the jEdit dataset (Dit et al. 2013e) and to illustrate that the implemented technique produces the same results as the original technique.

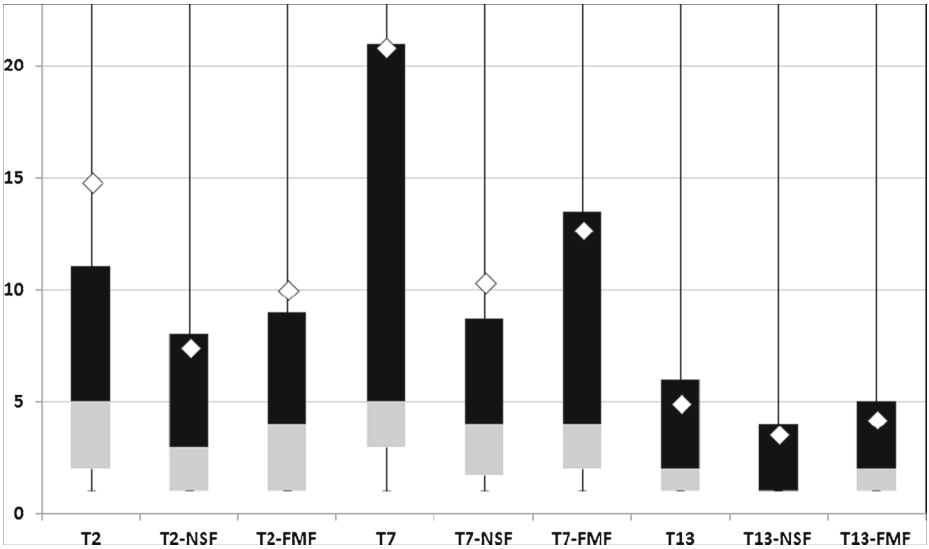

Fig. 9 *Box* plots of effectiveness measure (see y-axis) comparing (*i*) the techniques that produced the best results in (Dit et al. 2013e) for the jEdit4.3 dataset (e.g., T_2, T_7, and T_{13} - see Fig. 8 for exact names) against the (*ii*) No Scenario Filter (suffix NSF) and (*iii*) Frequent Methods Filter (FMF)

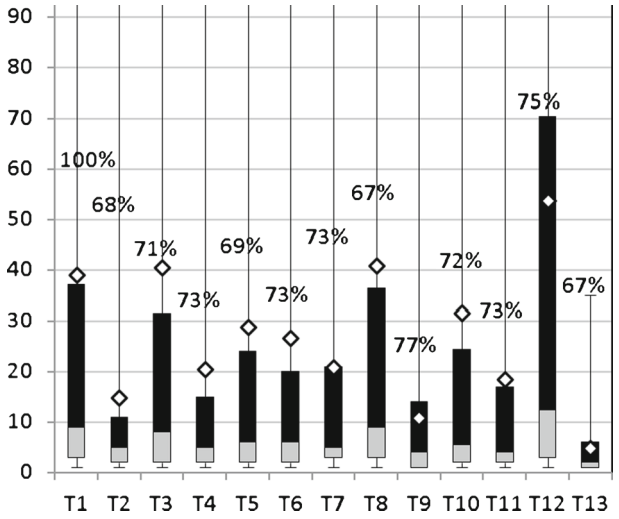

Fig. 10 This figure that was first published in (Dit et al. 2013e) Fig. 4(c) is reproduced in this paper for comparison purposes only with Fig. 8

6.2 Experimenting with New Ideas

Using the *IRDynWM* FLT (Dit et al. 2013e) as a starting point we experimented with incorporating new ideas for further improving the results. We describe the two new ideas and present their results in comparison with the original ones.

6.2.1 Filtering Frequent Methods from Execution Traces

The first idea that we instantiated in TraceLab consisted of filtering out some of the "noise" found in execution traces. More specifically, given a set of execution traces we identify the methods that appear in more than *X%* (i.e., a user defined threshold) of execution traces and we filter them out from the results produced by the *IRDynWM* technique. For example, consider our jEdit 4.3 dataset (Dit et al. 2013a, e) which contains 150 execution traces generated while exercising particular scenarios. Based on a specified threshold (e.g., 66 %) we (i) identified the methods that appear in 100 traces or more, and we (ii) filtered them out of the results produced by *IRDynWM*. Our intuition was that if a particular method captured in an execution trace appears in a large number of traces, the probability of that method to be part of a specific feature is low and therefore, could be eliminated. In a way, this filtering technique is similar to the process of eliminating stop words from corpora, where the stop words were identified as appearing frequently and carrying no real meaning in the corpus.

We implemented this idea based on the existing *IRDynWM*, which resulted in the $IRDynWM_{FrequentMethodFilter}$ or $IRDynWM_{FMF}$ technique (see the bottom part highlighted with a red rectangle in Fig. 7). In other words, the implementation of the *IRDynWM* technique requires all the components that are outside the highlighted area of Fig. 7, whereas the implementation of $IRDynWM_{FMF}$ requires all the components, including the ones found in the highlighted area of Fig. 7). The implementation required the following steps. First, we added two new components to (i) examine all the execution traces from the dataset (component *Load All Execution Traces*) and (ii) identify the methods that appear in more than *X%* of

traces, with *X%* being the threshold specified by the user (component *Compute Method Frequencies*). Second, for each query in the while loop we instantiated the same component three times to filter out the most frequent execution trace methods from each technique in the original experiment. For example, the results produced by the *IRDynPageRank* FLT were used as input for the $IRDynPageRank_{FMF}$, which filtered the frequent methods produced by the *IRDynPageRank* component.

The results produced by the $IRDynWM_{FMF}$ technique are presented in Section 6.2.3, and are compared against the results produced by the *IRDynWM* technique.

6.2.2 Filtering "No Scenario" Methods from a Trace

In case a large set of execution traces is not available (i.e., the prerequisite for $IRDynWM_{FMF}$ is not satisfied), a developer can use only one execution trace to get improved results, by collecting an execution trace that exercises *no scenario* (i.e., without exercising any specific features of the software). The execution trace was collected from the moment the application started to the moment the application terminated, without exercising any user features in the meantime. The methods captured in the *No Scenario* trace were filtered from the results produced by *IRDynWM*, resulting in the $IRDynWM_{NoScenarioFilter}$ *or* $IRDynWM_{NSF}$ technique. The intuition behind this idea is that the *No Scenario* trace contains a number of methods that are not associated with any specific scenario (i.e., generic methods), which can be filtered in order to improve the results.

The implementation of this technique is similar to the one presented in Fig. 7. The major modification was that the *Load All Execution Traces* and *Compute Method Frequencies* components were replaced with a component that loaded a user-specified *no scenario* execution trace and extracted the methods that will be filtered. In addition, the $IRDynWM_{FMF}$ composite nodes were replaced with corresponding $IRDynWM_{NSF}$ composite nodes.

The results generated by the $IRDynWM_{NSF}$ technique are presented in Section 6.2.3, and are compared against the results produced by the *IRDynWM* and $IRDynWM_{FMF}$ techniques.

6.2.3 Results for Evaluating the New Ideas

Figure 9 shows side by side the box plots of the effectiveness measure produced by *IRDynWM*, $IRDynWM_{FMF}$ and $IRDynWM_{NSF}$. For the comparison, we choose the best three configurations of PageRank, HITS Authority and HITS Hubs that produced the best results for the jEdit dataset in (Dit et al. 2013e), which are T_2, T_7 and T_{13} (see Fig. 8 for the labels). A

Table 5 Descriptive statistics for the box plots presented in Fig. 9. The first row (Percentage Features) represents the percentage of features for which the technique was able to locate at least one relevant method

	T_2	T_2-NSF	T_2-FMF	T_7	T_7-NSF	T_7-FMF	T_{13}	T_{13}-NSF	T_{13}-FMF
Percentage features	68 %	59 %	64 %	73 %	59 %	68 %	67 %	59 %	66 %
Min	1	1	1	1	1	1	1	1	1
25th	2	1	1	3	1.75	2	1	1	1
Median	5	3	4	5	4	4	2	1	2
75th	11	8	9	21	8.75	13.5	6	4	5
Max	237	81	145	170	141	142	35	40	35
Mean	14.81	7.47	10.02	20.85	10.36	12.72	4.92	3.56	4.25
Standard deviation	34.24	11.71	21.18	32.98	19.68	21.45	6.19	5.96	5.51

complementary view of Fig. 9 is given by Table 5, which contains descriptive statistics of the box plots generated by those techniques.

Figure 9 and Table 5 show that the $IRDynWM_{FMF}$ (e.g., T_2-*FMF*, T_7-*FMF* and T_{13}-*FMF*) techniques generate better results in terms of the effectiveness measure than *IRDynWM*, and that $IRDynWM_{NSF}$ produces better results than $IRDynWM_{FMF}$. For example, for T_2, the median value was five, whereas for T_2-FMF and T_2-NSF the median values for four and three respectively. The same trend is observed for the average values: 14.81, 10.02 and 7.47 for T_2, T_2-FMF and T_2-NSF, respectively.

Our two experimental ideas produced better results than the best results presented in (Dit et al. 2013e) for the jEdit dataset. However, the improvement in "precision", comes at the cost of potentially filtering out relevant methods. For example in Table 5 row *Percentage Features* shows the percentage of features for which that particular technique was able to identify at least one relevant method. As the table indicates, filtering additional methods removes noise (i.e., irrelevant methods to the feature), as well as some relevant methods.

7 Discussion

Although for this particular dataset the two experimental ideas produced better results than the ones reported in (Dit et al. 2013e), there is still more research to be done (e.g., investigate the impact of removing also relevant methods, automatically setting the threshold for $IRDynWM_{FMF}$, ensuring generalizability, considering more advanced techniques for analyzing traces (Egyed 2003; Eisenbarth et al. 2001), etc.) before considering these ideas as viable techniques. However, this is beyond the scope of this paper.

The main goal of these examples was to illustrate the support that our *Component Library* and the TraceLab framework can offer to researchers, who can quickly test new ideas and get some preliminary results to assess the feasibility of those ideas, and decide if it is worth pursuing them or not.

7.1 Reproducing Traceability Link Recovery Experiments

Oliveto et al. (Oliveto et al. 2010) investigated the equivalence of results produced for the traceability link recovery task using different IR techniques, such as VSM (Salton et al. 1975), LSI (Deerwester et al. 1990), JS (Abadi et al. 2008), and a topic modeling technique called Latent Dirichlet Allocation (LDA) (Blei et al. 2003). The evaluation was performed on two datasets, EasyClinic and eTour (Oliveto et al. 2010) and they reported the precision and recall of the results in addition to link overlap metrics. They performed Principal Component Analysis (PCA) to determine which techniques are equivalent in terms of performance in traceability link recovery. The authors found that VSM, LSI, and JS are equivalent, while LDA provides orthogonal results.

The approach proposed by Oliveto et al. is another example from our mapping study that can be entirely reproduced in TraceLab. The TraceLab experiment includes the same datasets used in the original evaluation and the same settings. Figure 11 shows the experiment in TraceLab and Fig. 12 shows the precision and recall curves obtained from reproducing the experiment performed by Oliveto et al.. The components on top perform standard preprocessing techniques on the source and target artifacts. The four components in the center (*Vector Space Model*, *Jensen-Shannon divergence*, *Latent Semantic Analysis* and *Latent Dirichlet Analysis*) correspond to the four IR techniques used for comparison. The *Principal Component Analysis* component is responsible for determining which of the four IR techniques produce

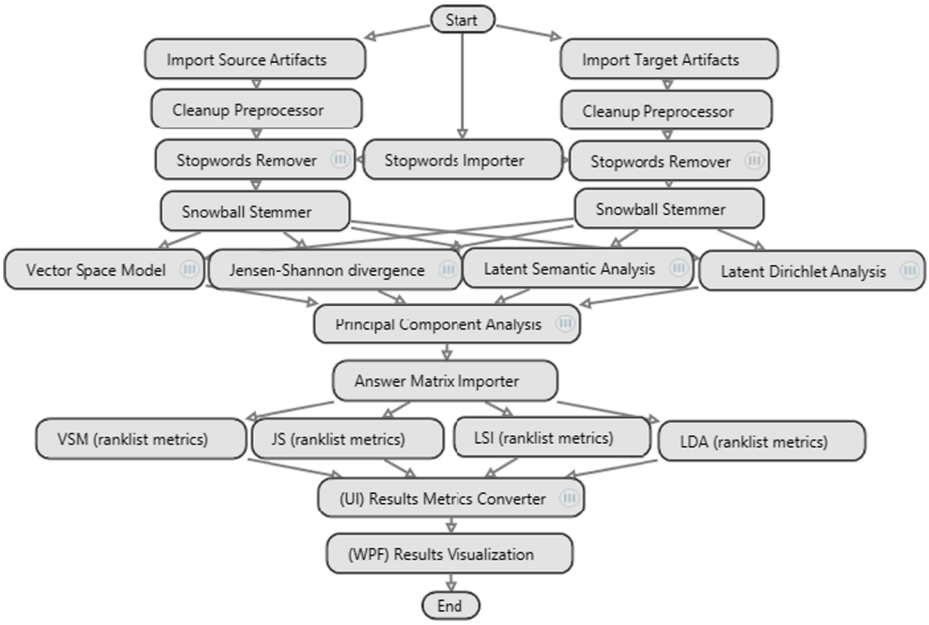

Fig. 11 TraceLab experiment for reproducing the experiment proposed by Oliveto et al. (2010)

equivalent results, and which produce orthogonal (complementary) results. The bottom part of the experiment provides the standard functionality for computing results for traceability link recovery, namely, importing the oracle, computing the precision and recall results and visualizing them.

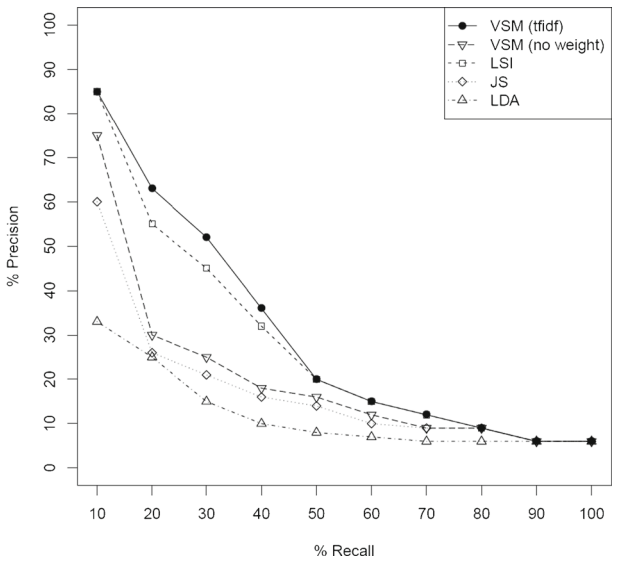

Fig. 12 Precision and recall curves obtained from reproducing in TraceLab the experiments in (Oliveto et al. 2010)

Based on the findings of Oliveto et al. (2010), Gethers et al. (2011) investigated the effects of combining orthogonal IR techniques to support traceability link recovery. In addition to using VSM and JS IR techniques, the authors also used the Relational Topic Model (RTM) (Chang and Blei 2010), which takes into account links between artifacts when determining the topics related to those artifacts.

Their approach implemented PCA to determine the level of contribution of each technique, which is then used as a λ parameter for an affine transformation between pairs of techniques. The authors performed an evaluation on four datasets (e.g., EAnci, eTour, EasyClinic, and SMOS – see (Oliveto et al. 2010)) and they reported precision, recall, and average precision of the results as well as link overlap metrics. Based on the results of their evaluation, the authors confirmed that VSM and JS produce equivalent results, and that RTM produces orthogonal results. The authors also found that using the hybrid approach which ranks artifacts based on a combination of scores produced by VSM and RTM (e.g., VSM + RTM) and the hybrid approach that combines JS and RTM (e.g., JS + RTM) significantly increases the accuracy of traceability link recovery, as compared to standalone techniques (e.g., either VSM, JS, or RTM without any combination).

The approach proposed by Gethers et al. (2011) is another example identified in our mapping study, which can be reproduced in TraceLab (see Fig. 13). This experiment reused many components from the *Component Library* that were also used in the previously mentioned traceability experiment by Oliveto et al. (see Fig. 11). These components are related to preprocessing, VSM, JS, PCA, for computing the precision and recall values for various configurations as well as the UI component for displaying the results. The components that are found only in this experiment (which were also imported from the *Component Library*) are the

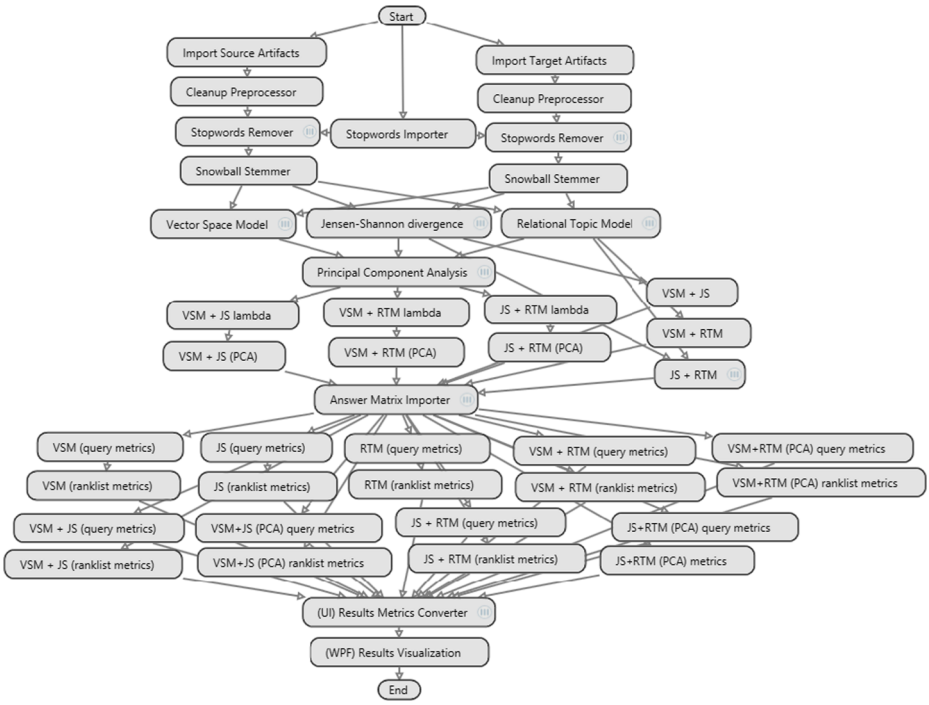

Fig. 13 TraceLab experiment for reproducing the experiment proposed by Gethers et al. (2011)

ones that use affine transformation to combine two existing techniques in order to generate a new one. These components are the ones with the name $IRTechnique_1$ + $IRTechnique_2$ (e.g., *VSM + JS*, *VSM + RTM*, etc.)

8 TraceLab: Alternative Uses

The community surrounding *TraceLab*, the *Component Development Kit* and the *Component Library* enables the replication of experiments, helps new researchers to become productive more quickly, and encourages innovation through equipping researchers with the means to synthesize techniques and to rigorously explore and evaluate new ideas across different domain. In addition, we argue that the curriculum for software engineering classes could be positively impacted as well.

First, class assignments and projects could be designed to support their submission using TraceLab. Students can be given a partial TraceLab experiment (e.g., loading data, saving data), and could be asked to implement one or more components that compute the results for a technique that supports a maintenance task. Alternatively, students can be given a complete experiment (which implements a technique for traceability link recovery, feature location, or any other maintenance tasks) and asked to improve on the existing techniques by writing new components, and compare the results to the original technique (which can be used as a baseline). The benefit of using TraceLab is that (i) it provides students the flexibility to implement code in some of the most popular languages (Java, C++, C#) on any OS (e.g., Windows, Linux, Mac), and it provides the instructor with a uniform method for evaluating and grading the assignments or projects (i.e., the instructor has to run an experiment that will be used as a ground truth alongside the experiment submitted by the student).

Second, class assignments or projects can be assigned to students as part of a customized *Challenge*, using *TraceLab's challenge* feature. Instructors can define the challenge (i.e., the problem), provide the necessary datasets, metrics, an experimental harness (in the form of a TraceLab experiment) in which student's solutions can be "plugged", and ask students to implement their solutions and submit them to an online portal supporting leaderboards, where challenge results will be compared and ranked. In the research community, one of these TraceLab Challenges made its debut during the 7th Traceability of Emerging Forms of Software Engineering (TEFSE) workshop in 2013, and will continue for the TEFSE workshops in 2015 and 2017.

9 Limitations

This section discusses some potential limitations for conducting research using TraceLab, the *Component Development Kit* and the *Component Library.*

For example, the current infrastructure does not support collecting metrics (e.g., LOC, cyclomatic complexity, depth of inheritance tree, coupling between objects, etc.) from different software systems. However, these metrics could be computed with external tools and imported into TraceLab and used in experiments.

Running experiments in TraceLab from code hosted in a .NET process is in general slower than running the code associated with experiments natively. For example, for typical experiments the code would require a few seconds more to run on TraceLab, but for computationally expensive experiments (which could take hours, or days on native code) TraceLab experiments could take from minutes to hours longer to run than experiments ran in native code. Therefore, the time or speed factors in evaluating an approach would need to be considered.

We attempted to identify papers that covered a number of topics in SM, which we were familiar with or had expertise with. Within the papers we covered, in some cases we were unable to obtain exact implementations due to lack of specific details or availability of tools. Additionally, many experiments cannot be reproduced directly because the datasets under study were undisclosed or unavailable.

The *CL* and *CDK* do not implement every technique and building block found in the mapping study. The amount of time, manpower, and testing required to do so would be far beyond the resources available. That being said, we tried to implement as many of the techniques that we could in order to show the efficacy and usefulness of TraceLab as a research tool in the domain of software evolution and maintenance. We are continuously working on driving new research projects with TraceLab (Cleland-Huang et al. 2013; Dekhtyar and Hilton 2013; Alhindawi et al. 2013; Hays et al. 2013; Li and Hayes 2013; Rempel et al. 2013; Dit et al. 2013c; Wieloch et al. 2013) and encourage others to do so as well.

A major issue that prevented us from using or implementing certain tools was their copyright licensing. In some cases they do not use permissive licenses, and even if the source code was available its license did not permit distribution. TraceLab is released under the open source license GPL, which we follow as well with the *CL* and *CDK*. Developers may release their own components under any license they wish, but if they wish to extend or modify the *CL* or *CDK*, they must also release under GPL.

With any new technology or framework, as is the case with TraceLab and the *CL*/*CDK*, there is an inherent learning curve that needs to be overcome before researchers can take advantage of this infrastructure in order to support their research and contribute to the community as well. To facilitate this learning process and to make it easier for new users to create new experiments and components, we provide numerous online tutorials (e.g., wikis and videos) to help new users get started. In fact, most of our early adopters have viewed a selection of tutorials prior to adoption.

10 Conclusions and Future Work

This paper is an extension of our previous work (Dit et al. 2013b) and addressed the *reproducibility* problem associated with experiments in SM research. Our goal was to support and accelerate research in SE by providing a body of actionable knowledge in the form of reproduced experiments and a *Component Library* and *Component Development Kit* that can be used as the basis to generate novel, and most importantly reproducible techniques.

After conducting a mapping study of SM techniques in the areas of traceability link recovery, feature location, program comprehension and duplicate bug report detection, we identified 27 papers and techniques that we used to generate a library of TraceLab components. We implemented a subset of these techniques as TraceLab experiments to illustrate TraceLab's potential as a research framework and to provide a basis for implementing new techniques.

It is obvious that our effort does not cover the entire range of SM papers or techniques. Therefore, in the future, we are determined to continually expand the TraceLab *Component Library* and *Development Kit* by including more techniques and expanding it to other areas of SM (e.g., impact analysis, developer recommendation, software categorization, etc.). In addition, we are expanding our online tutorials to make it easier for newcomers to get started with TraceLab, and we encourage other researchers to contribute to this body of knowledge for the benefit of conducting reproducible research, which in turn, will benefit the entire research community.

Acknowledgments This work is supported in part by the United States NSF CNS-0959924, NSF CCF-1218129, and NSF CCF-1016868 grants. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors. We would also like to acknowledge the team of researchers from DePaul University led by Jane Cleland-Huang: Ed Keenan, Adam Czauderna, Greg Leach, and Piotr Pruski. This work would not have been possible without their continuous support on the TraceLab project.

References

Abadi A, Nisenson M, Simionovici Y (2008) A traceability technique for specifications. In: 16th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, The Netherlands. pp 103–112

Alhindawi N, Meqdadi O, Bartman B, Maletic JI (2013) A tracelab-based solution for identifying traceability links using LSI. In: International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13). pp 79-82

Asuncion H, Asuncion A, Taylor R (2010) Software traceability with topic modeling. In: 32nd International Conference on Software Engineering (ICSE'10)

Barr E, Bird C, Hyatt E, Menzies T, Robles G (2010) On the shoulders of giants. In: FSE/SDP Workshop on Future of Software Engineering Research (FoSER'10), Santa Fe, New Mexico, USA, ACM, 1882368, pp 23–28. doi:10.1145/1882362.1882368

Blei DM, Ng AY, Jordan MI (2003) Latent Dirichlet Allocation. J Mach Learn Res 3:993–1022

Borg M, Runeson P, Ardö A (2013) Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. Empir Softw Eng (EMSE):1–52. doi:10.1007/s10664-013-9255-y

Brin S, Page L (1998) The anatomy of a large-scale hypertextual web search engine. In: 7th International Conference on World Wide Web, Brisbane, Australia. pp 107–117

Capobianco G, De Lucia A, Oliveto R, Panichella A, Panichella S (2009) On the role of the nouns in IR-based traceability recovery. In: 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, British Columbia, Canada, May 17–19. pp 148–157

Chang J, Blei DM (2010) Hierarchical relational models for document networks. Statistics, Annals of Applied

Chen X, Hosking J, Grundy J (2011) A combination approach for enhancing automated traceability In: 33rd IEEE/ACM International Conference on Software Engineering (ICSE'11), NIER Track, Honolulu, Hawaii, USA, May 21–28. ACM, 1985943, pp 912–915. doi:10.1145/1985793.1985943

Cleland-Huang J, Czauderna A, Dekhtyar A, O. G, Huffman Hayes J, Keenan E, Leach G, Maletic J, Poshyvanyk D, Shin Y, Zisman A, Antoniol G, Berenbach B, Egyed A, Maeder P (2011) Grand challenges, benchmarks, and TraceLab: developing infrastructure for the software traceability research community. In: 6th ICSE2011 International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE2011), Honolulu, HI, USA, May 23

Cleland-Huang J, Shin Y, Keenan E, Czauderna A, Leach G, Moritz E, Gethers M, Poshyvanyk D, Hayes JH, Li W (2012) Toward actionable, broadly accessible contests in software engineering. In: 34th IEEE/ACM International Conference on Software Engineering (ICSE'12), New Ideas and Emerging Results Track, Zurich, Switzerland, June 2–9. pp 1329–1332

Cleland-Huang J, Mirakhorli M, Czauderna A, Wieloch M (2013) Decision-Centric Traceability of architectural concerns. In: International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13). pp 5–11

D'Ambros M, Lanza M, Robbes R (2012) Evaluating Defect Prediction Approaches: a Benchmark and an Extensive Comparison. Empir Softw Eng (ESE) 17(4–5):531–577. doi:10.1007/s10664-011-9173-9

De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S (2011) Improving IR-based traceability recovery using smoothing filters. In: 19th IEEE International Conference on Program Comprehension (ICPC'11), Kingston, Ontario, Canada, June 22–24. IEEE, pp 21–30

De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S (2012) Using IR methods for labeling source code artifacts: is it worthwhile? In: 20th IEEE International Conference on Program Comprehension (ICPC'12), Passau, Germany. pp 193–202

Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R (1990) Indexing by Latent Semantic Analysis. J Am Soc Inf Sci 41(6):391–407

Dekhtyar A, Hilton M (2013) Human recoverability index: a TraceLab experiment. In: International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13). pp 37–43

Dit B, Guerrouj L, Poshyvanyk D, Antoniol G (2011) Can better identifier splitting techniques help feature location? In: 19th IEEE International Conference on Program Comprehension (ICPC'11), Kingston, Ontario, Canada, June 22–24. pp 11–20

Dit B, Moritz E, Poshyvanyk D (2012) A TraceLab-based solution for creating, conducting, and sharing feature location experiments. In: 20th IEEE International Conference on Program Comprehension (ICPC'12), Passau, Germany, June 11–13. pp 203–208

Dit B, Holtzhauer A, Poshyvanyk D, Kagdi H (2013a) A Dataset from change history to support evaluation of software maintenance tasks. In: 10th Working Conference on Mining Software Repositories (MSR'13), Data Track, San Francisco, CA, May 18–19. pp 131–134

Dit B, Moritz E, Linares-Vásquez M, Poshyvanyk D (2013b) Supporting and accelerating reproducible research in software maintenance using TraceLab component library. In: 29th IEEE International Conference on Software Maintenance (ICSM'13), Eindhoven, the Netherlands, September 22–28. pp 330–339

Dit B, Panichella A, Moritz E, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A (2013c) Configuring topic models for software engineering tasks in TraceLab. In: 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13), San Francisco, California, May 19. pp 105–109

Dit B, Revelle M, Gethers M, Poshyvanyk D (2013d) Feature Location in Source Code: A Taxonomy and Survey. J Softw: Evol Process (JSEP) 25(1):53–95. doi:10.1002/smr.567

Dit B, Revelle M, Poshyvanyk D (2013e) Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software. Empir Softw Eng 18(2):277–309. doi:10.1007/s10664-011-9194-4

Do H, Elbaum S, Rothermel G (2005) Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Empir Softw Eng 10(4):405–435

Egyed A (2003) A scenario-driven approach to trace dependency analysis. IEEE Trans Softw Eng (TSE) 29(2):116–132

Eisenbarth T, Koschke R, Simon D (2001) Feature-driven program understanding using concept analysis of execution traces. Paper presented at the IWPC

Enslen E, Hill E, Pollock L, Vijay-Shanker K (2009) Mining source code to automatically split identifiers for software analysis. In: 6th IEEE Working Conference on Mining Software Repositories (MSR'09), Vancouver, BC, Canada May 16–17. pp 71–80

FETCH (2014) (Fact Extraction Tool CHain) University of Antwerp. http://lore.ua.ac.be/fetchWiki/. Accessed 15 April 2014

Gay G, Haiduc S, Marcus M, Menzies T (2009) On the use of relevance feedback in IR-based concept location. In: 25th IEEE International Conference on Software Maintenance (ICSM'09), Edmonton, Canada, September. pp 351–360

Gethers M, Oliveto R, Poshyvanyk D, De Lucia A (2011) On integrating orthogonal information retrieval methods to improve traceability link recovery. In: 27th IEEE International Conference on Software Maintenance (ICSM'11), Williamsburg, Virginia, USA, September 25–30. pp 133–142

González-Barahona JM, Robles G (2012) On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. Empir Softw Eng (ESE) 17(1–2):75–89. doi:10.1007/s10664-011-9181-9

Haiduc S, Aponte J, Marcus A (2010) Supporting program comprehension with source code summarization. In: 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), Cape Town, South Africa. ACM, 1810335, pp 223–226. doi:10.1145/1810295.1810335

Hays M, Hayes JH, Stromberg AJ, Bathke AC (2013) Statistical analysis for traceability experiments: Software verification and validation research laboratory (SVVRL) of the University of Kentucky. In: International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13). pp 90–94

Jørgensen M (2004) A review of studies on expert estimation of software development effort. J Syst Softw (JSS) 70(1):37–60

Kaushik N, Tahvildari L (2012) A comparative study of the performance of IR models on duplicate bug detection. In: 16th European Conference on Software Maintenance and Reengineering (CSMR'12). IEEE Computer Society, 2192561, pp 159–168. doi:10.1109/csmr.2012.78

Keenan E, Czauderna A, Leach G, Cleland-Huang J, Shin Y, Moritz E, Gethers M, Poshyvanyk D, Maletic J, Hayes JH, Dekhtyar A, Manukian D, Hussein S, Hearn D (2012) TraceLab: an experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In: 34th IEEE/ACM International Conference on Software Engineering (ICSE'12), Zurich, Switzerland, June 2–9. pp 1375–1378

Kepler (2013) The Kepler Project - University of California.https://kepler-project.org/. Accessed 15 April 2014

Kitchenham BA, Budgen D, Brereton OP (2011) Using Mapping Studies as the Basis for Further Research - A Participant-Observer Case Study. Inf Softw Technol 53(6):638–651. doi:10.1016/j.infsof.2010.12.011

Kleinberg JM (1999) Authoritative Sources in a Hyperlinked Environment. J ACM 46(5):604–632

Li W, Hayes JH (2013) Query+ enhancement for semantic tracing (QuEST): Software verification and validation research laboratory (SVVRL) of the University of Kentucky. In: International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13). pp 95–99

Liu D, Marcus A, Poshyvanyk D, Rajlich V (2007) Feature location via information retrieval based filtering of a single scenario execution trace. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), Atlanta, Georgia, November 5–9. pp 234–243

Marcus A, Sergeyev A, Rajlich V, Maletic J (2004) An information retrieval approach to concept location in source code. In: 11th IEEE Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, November 9–12. pp 214–223

MathWorks (2013) Matlab http://www.mathworks.com/products/matlab/. Accessed 15 April 2014

MathWorks (2013) Simulink http://www.mathworks.com/products/simulink/. Accessed 15 April 2014

Menzies T, Caglayan B, Kocaguneli E, Krall J, Peters F, Turhan B (2012) The PROMISE repository of empirical software engineering data. http://promisedata.googlecode.com

Mytkowicz T, Diwan A, Hauswirth M, Sweeney P (2010) The effect of omitted-variable bias on the evaluation of compiler optimizations. IEEE Comput 43(9):62–67. doi:10.1109/mc.2010.214

Oliveto R, Gethers M, Poshyvanyk D, De Lucia A (2010) On the equivalence of information retrieval methods for automated traceability link recovery. In: 18th IEEE International Conference on Program Comprehension (ICPC'10), Braga, Portugal, June 30 - July 2. pp 68–71

Panichella A, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A (2013a) How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms. In: 35th IEEE/ACM International Conference on Software Engineering (ICSE'13), San Francisco, CA, May 18–26. pp 522–531

Panichella A, McMillan C, Moritz E, Palmieri D, Oliveto R, Poshyvanyk D, De Lucia A (2013b) When and how using structural information to improve IR-based traceability recovery. In: 17th European Conference on Software Mainenance and Reengineering (CSMR'13), Genova, Italy, March 5–8. pp 199–208

Petersen K, Feldt R, Mujtaba S, Mattsson M (2008) Systematic mapping studies in software engineering. Paper presented at the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08), Italy

Poshyvanyk D, Guéhéneuc YG, Marcus A, Antoniol G, Rajlich V (2007) Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans Softw Eng (TSE) 33(6):420–432

Rapid-I (2013) Rapid Miner http://rapid-i.com/content/view/181/190/. Accessed 15 April 2014

Rempel P, Mader P, Kuschke T (2013) Towards feature-aware retrieval of refinement traces. In: International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13). pp 100–104

Revelle M, Poshyvanyk D (2009) An exploratory study on assessing feature location techniques. In: 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, British Columbia, Canada, May 17–19. pp 218–222

Robles G (2010) Replicating MSR: a study of the potential replicability of papers published in the mining software repositories proceedings. In: 7th IEEE Working Conference on Mining Software Repositories (MSR'10), Cape Town, South Africa, May 2–3. pp 171–180. doi:10.1109/msr.2010.5463348

R-Project (2013) R http://www.r-project.org/. Accessed 15 April 2014

Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: 29th IEEE/ACM International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, May 20–26. pp 499–510

Salton G, Wong A, Yang CS (1975) A vector space model for automatic indexing. Commun ACM (CACM) 18(11):613–620

Sayyad SJ, Menzies TJ (2005) The PROMISE repository of software engineering databases. http://promise.site.uottawa.ca/SERepository. Accessed July 17 2007

Scanniello G, Marcus A (2011) Clustering support for static concept location in source code. In: 19th IEEE International Conference on Program Comprehension (ICPC'11), Kingston, Ontario, Canada, June 22–24. pp 1–10

Sheffield TUo (2011) GATE: general architecture for text engineering. http://gate.ac.uk/. Accessed April 24 2013

Shull FJ, Carver JC, Vegas S, Juristo N (2008) The role of replications in Empirical Software Engineering. Empir Softw Eng 13(2):211–218. doi:10.1007/s10664-008-9060-1

Taverna, (2014) myGrid team http://www.taverna.org.uk/. Accessed 15 April 2014

Tian K, Revelle M, Poshyvanyk D (2009) Using latent Dirichlet allocation for automatic categorization of software. In: 6th IEEE Working Conference on Mining Software Repositories (MSR'09), Vancouver, British Columbia, Canada, May 16–17. pp 163–166

Waikato TUo (2013) WEKA http://www.cs.waikato.ac.nz/ml/weka/. Accessed 15 April 2014

Wang X, Zhang L, Xie T, Anvik J, Sun J (2008) An approach to detecting duplicate bug reports using natural language and execution information. In: 30th IEEE/ACM International Conference on Software Engineering (ICSE'08), Leipzig, Germany, May 10–18. pp 461–470

Wieloch M, Amornborvornwong S, Cleland-Huang J (2013) Trace-by-classification: a machine learning approach to generate trace links for frequently occurring software artifacts. In: International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13). pp 110–114

Wiese A, Ho V, Hill E (2011) A Comparison of stemmers on source code identifiers for software search. In: 27th IEEE International Conference on Software Maintenance (ICSM'11), Williamsburg, Virginia, USA, September 25–30. pp 496–499

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: 3rd International Workshop on Predictor Models in Software Engineering (PROMISE'07), Minneapolis, MN, USA, May 19–20. IEEE, p 9

Bogdan Dit is a Ph.D. Candidate in the Computer Science Department at the College of William and Mary where he is a member of SEMERU research group. He is advised by Dr. Denys Poshyvanyk. Bogdan obtained his B.S. in Computer Science from Babeş-Bolyai University, Romania and his M.S. in Computer Science from Wayne State University. His research interests include software engineering, software maintenance and evolution, feature location, application of information retrieval in software engineering, mining of software repositories, reproducibility of experiments in software maintenance, and program comprehension. He is a student member of the IEEE and ACM.

Evan Moritz received his M.S. degree in Computer Science from the College of William and Mary in Williamsburg, Virginia in 2013, where he was a member of the SEMERU research group. He was advised by Dr. Denys Poshyvanyk. Evan also obtained his B.S. from the College of William and Mary in 2011. His research interests include software engineering, software maintenance and evolution, and reproducibility of experiments in software engineering.

Mario Linares-Vásquez is a Ph.D. student at the College of William and Mary advised by Dr. Denys Poshyvanyk. He received his B.S. in Systems Engineering from Universidad Nacional de Colombia in 2005, and his M.S. in Systems Engineering and Computing from Universidad Nacional de Colombia in 2009. His research interests include software evolution and maintenance, software reuse, mining software repositories, application of data mining and machine learning techniques to support software engineering tasks. He is a student member of the IEEE and ACM.

Denys Poshyvanyk is an Associate Professor at the College of William and Mary in Virginia. He received his Ph.D. degree in Computer Science from Wayne State University in 2008. He also obtained his M.S. and M.A. degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine and Wayne State University in 2003 and 2006, respectively. His research interests are in software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis and metrics. He is a member of the IEEE and ACM.

Jane Cleland-Huang is a Professor at DePaul University's School of Computing. She received a Ph.D. in Computer Science from the University of Illinois at Chicago. Her research interests include requirements traceability with emphasis on tracing non-functional requirements (NFRs) across the system lifecycle. She is a member of the IEEE Computer Society and the Software Engineering Research Consortium (SERC), and serves as regional director of the Center of Excellence for Software Traceability. Her work is currently funded by grants from the National Science Foundation and Siemens Corporate Research.