# Supporting and Accelerating Reproducible Research in Software Maintenance using TraceLab Component Library

Bogdan Dit, Evan Moritz, Mario Linares-Vásquez, and Denys Poshyvanyk
Computer Science Department
The College of William and Mary
Williamsburg, VA, USA
{bdit, eamoritz, mlinarev, denys}@cs.wm.edu

***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 lot 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 mapping study of a number of representative maintenance techniques and approaches and implemented them as a library of experiments and 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 would (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, which produce improved results.**

***Keywords—software maintenance, reproducible, experiments, case studies, TraceLab***

## I. 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 example, the 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 [1, 2, 3, 4, 5, 6].

For example, a survey on feature location (FL) techniques by Dit *et al.* [1] revealed that only 5% of the papers surveyed (three out of 60 papers) used in their evaluation the same dataset that was used in evaluating other techniques, and that only 38% of the papers surveyed (23 out of 60 papers) compared their proposed feature location technique against a small number of previously introduced feature location techniques. In addition, these findings are consistent with the ones from the study by Robles [2], 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 [6] identified the factors affecting the reproducibility of results in empirical software engineering research and proposed a methodology for determining the reproducibility of a study. In another study, Mytkowicz *et al.* [3] 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.* [4] argued that many approaches in bug prediction have not been evaluated properly (*i.e.*, they were either evaluated by themselves, or they were compared against a limited set of other approaches), and highlight 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 [1, 2, 3, 4, 5, 6], 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 [7, 8], Eclipse Bug Data [9], SEMERU feature location dataset [1], Bug Prediction Dataset [4], SIR [10], and others. In addition, different infrastructures for running experiments were introduced, such as TraceLab [11, 12, 13], RapidMiner [14], Simulink [15], Kepler [16], and others. However, among these, the most suitable framework for facilitating and advancing research in software engineering and maintenance is TraceLab (see Section III.B for an in-depth comparison and discussion of TraceLab's features with other tools). TraceLab is a plug-and-play framework that was specifically designed for facilitating creating, evaluating, comparing, and sharing experiments in software engineering and maintenance. 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*. We analyzed the approaches presented in 27

research papers and we implemented them as TraceLab experiments. In order to implement these SM approaches, we 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 existing SM techniques, and to exemplify how these components and experiments can 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 IV) to identify the set of techniques that we reproduced 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 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*, as well as using the 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: www.cs.wm.edu/semeru/TraceLab_CDK

The paper is organized as follows. Section II presents a motivating example that shows variability in results of applying a simple SM technique and challenges of reproducing those results without complete details. Section III introduces background details about TraceLab and presents a comparison with other tools. Section IV presents the mapping study performed, which we used to implement the Component Library and Development Kit (Section V). Section VI shows an example of reproducing an existing FL technique and presents details on improving it. Finally, Section VII discusses some potential limitations and Section VIII concludes the paper and introduces some ideas for future work.

## II. 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 [1, 2].

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 significant impact on the results.

To illustrate this point on a concrete example, we applied the popular IR technique Vector Space Model (VSM) [17] on the EasyClinic system from TEFSE 2009[1] challenge to recover

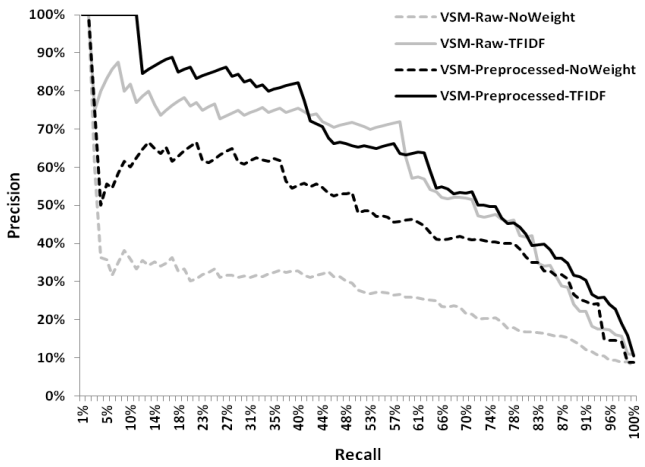


Figure 1 Precision-Recall curves for EasyClinic for recovering traceability links between use cases and classes using a VSM-based traceability technique and different preprocessing techniques (*raw* – gray color, *preprocessed* – black color) and weighting schemes (*no weight* – dash line, *tf-idf* –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

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.*, remove special characters, split identifiers and stem). The weighting schemes used were *no weighting* and *term frequency-inverse document frequency* (*tf-idf*) weighting. Figure 1 indicates the raw and basic preprocessing steps with gray and black color respectively, and the no weighting and tf-idf weighting with dashed line and solid line respectively. The results in Figure 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, such as one designed in TraceLab.

## III. 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

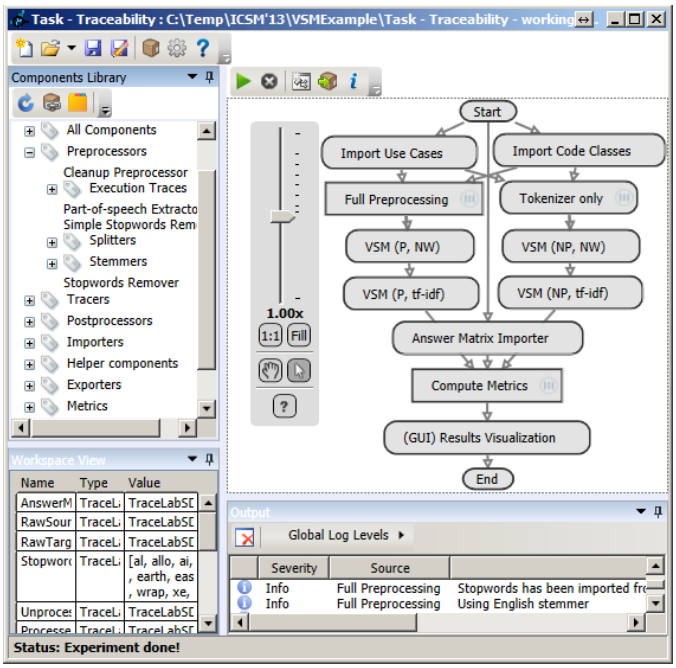


Figure 2 The four "quadrants" of TraceLab in clockwise order from top-right are (i) the sample TraceLab experiment that implements our motivating example in Section 2; (ii) an output window for reporting execution status of an experiment; (iii) the Workspace containing the data and the values of the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

### A. *TraceLab*

TraceLab [11, 12, 13] is a framework designed to support the reproducibility of experiments in software engineering and software maintenance (see Figure 2). More specifically, it 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 Kent State University, University of Kentucky, and the College of William and Mary.

The heart of a TraceLab experiment lies in its workflow of components and tools (see Figure 2 upper-right). An experiment is a collection of nodes (or components) connected in the form of a precedence graph. Each component communicates with the preceding and following nodes by storing and loading information to and from a common data-sharing interface called the *Workspace* (see Figure 2 lower-left). The status of an experiment is reported in the Output view (see Figure 2 lower-right). Individual components are engineered to implement a specific task and components that implement related tasks can be combined to form composite components, such as the node with rectangular edges labeled *Queries preprocessing* in Figure 4, which implements various tasks such as identifier splitting, stemming, and stopwords removal. In addition, TraceLab provides basic control flow within the experiment via decision nodes and *while loops* (see Figure 4).

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 Figure 2 upper-left). The *Component Library* will be included in the distribution of next official TraceLab release.

Table I 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]

| Tool / Feature | Yahoo pipes | Weka/ R. Miner | Simulink | Gate | Kepler | TraceLab |
|---|---|---|---|---|---|---|
| **GUI** | Y | Y | Y | N | Y | Y |
| **Type** | W | API, D | D | API, D | API, D | API, D |
| **License** | F | O | C | O | O | O |
| **Save/Load exp.** | Y | Y | Y | Y | Y | Y |
| **Composite comp.** | Y | N | Y | P | Y | Y |
| **Comp. Market** | Y | N | Y | Y | Y | Y |
| **Prog. Lang.** | - | Java | C/C++ Matlab. Fortran | Java | R C Matlab Java | Java R .NET lang. Matlab |
| **Platforms** | SAAS | W, L, M | W, L, M | W, L, M | W, L, M | W, L, M |

### B. *TraceLab Comparison with Other Tools*

There are also numerous other frameworks and tools that were 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 making 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 I compares TraceLab to some similar tools that also use a data-flow oriented GUI.

WEKA [18] 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 [14] 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.

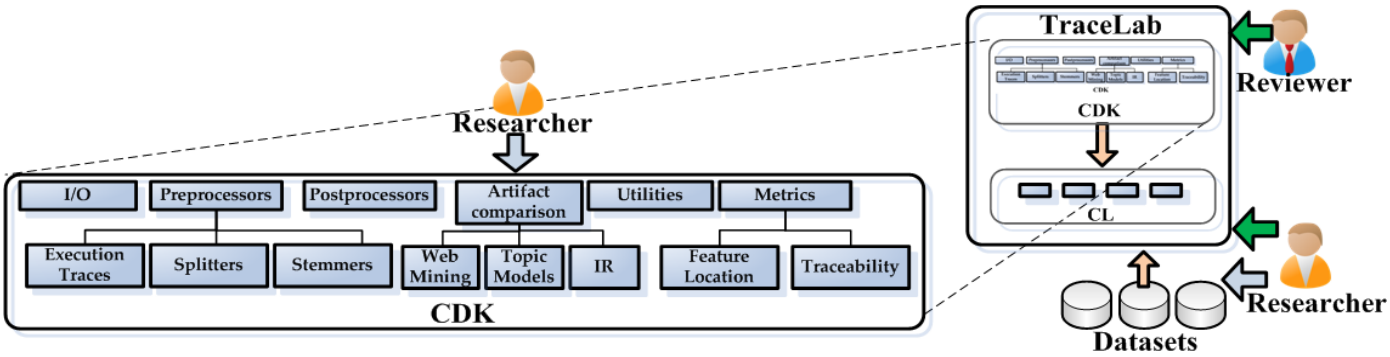


Figure 3 Diagram of the hierarchy of the *CDK* in the context of TraceLab. *CDK* and *CL* are part of TraceLab. Researchers can contribute to the *CDK* and the datasets (gray arrow), and reviewers and researchers (green arrow) can use TraceLab to verify details of existing experiments

Yahoo Pipes [19] is a data mashup tool with components for web retrieval, filtering and aggregation of web feeds, web pages, and other services. As the TraceLab composite components, pipes (*i.e.*, Yahoo pipes composite components) can be reused as building blocks for new pipes. In addition, pipes are shared/published through the Yahoo pipes website.

Simulink [15] 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.

GATE [20] 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 [16] 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.

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. TraceLab supports all major OS platforms (*e.g.*, Window, MacOS and Linux) and researchers can use Java, any .NET language (*e.g.*, C#, VB, C++), R or Matlab to implement their components.

## IV. Mapping Study of Software Maintenance Techniques

In this section we present the methodology, analysis and results of a mapping study [21] 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 V.A) and a *Component Library* (see Section V.B) that serves as a starting point for any interested researcher to implement new techniques or build upon existing ones.

For our study, we use the systematic mapping process described by Petersen *et al.* [22]. 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 the relevant ones, 4) classifying the papers, and 5) data extraction and generating the systematic map.

### *1) Defining the Research Questions*

Our goal is to identify a set of representative techniques from specific areas of SM, and use them to generate TraceLab components and experiments to accelerate and support research in SM. Therefore, our main guiding research question was formulated as: *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 where the authors have expertise, which allowed generating this initial body of knowledge that could support the research community, and one that we, and the research community, could contribute to, by constantly adding new techniques and components.

### *2) Conducting the Search*

In order to find these techniques, we narrowed the search space to the publications from the last ten years of a subset of journals and software engineering conferences (see our online appendix for more details). In addition, in our search we incorporated the "snowballing" discovery technique (*i.e.*, following references in the related work) discussed by Kitchenham *et al.* [21].

### *3) Screening Criteria*

The primary *inclusion* criterion consisted in 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

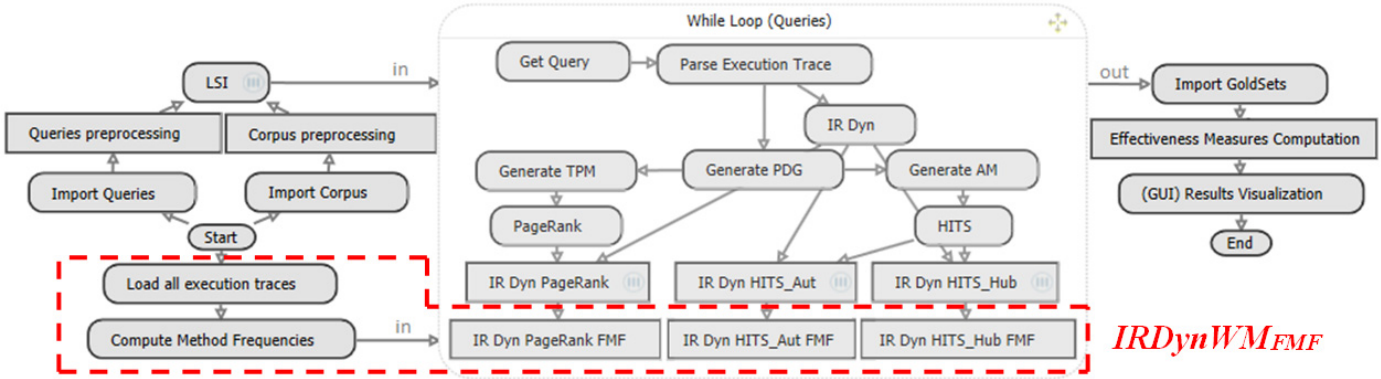


Figure 4 TraceLab experiment for reproducing the *IRDynWM* FLT [23] (without the components highlighted with red dashed border). TraceLab experiment for implementing $IRDynWM_{FMF}$ (all components, including the highlighted ones)

development time, or techniques that are outside the expertise of the authors. Third, we discarded techniques with numerous 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) Classification*

In our mapping study we used two independent levels of classification. The first one consisted of categorizing the papers based on the type of technique (*e.g.*, traceability link recovery, feature location, program comprehension and detecting duplicate bug reports) they presented (see Section IV.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, stopwords removal and others, were grouped under "preprocessing").

*5) Data extraction*

The list of papers that we identified in our study is presented in Table II in the first column along with the Google Scholar citation count as of August 9, 2013 (second column). The papers are grouped by their primary maintenance 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 V.B and Section VII).

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

## V. 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 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.

### A. 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. By providing these tools in a clear manner, the *Component Development Kit* facilitates the research evaluation process - 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 high-level tasks, such as I/O, preprocessing techniques, artifact comparison techniques, and metrics calculations (see Figure 3). Those levels 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.

Based on our 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.

### B. 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.

To register a component in TraceLab, a class must inherit from the *BaseComponent* abstract class in the TraceLab SDK. All components have a *Compute()* method which contains the desired functionality of the component within the context of a TraceLab experiment. Furthermore, all components have a component declaration attribute (or annotation in Java terminology) that describes information about the component,

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

| Technique Year / Venue / Name / Ref | Google Scholar Citation Count | Preprocessing | | | | | | | | | | Artifact Comparison | | | | | | | Metrics | | | Postprocessing | | | | Other |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 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 | 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** | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2008.ICPC.Abadi [24] | 50 | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | ✓ | ✓ | . | ✓ | . | . | . | ✓ | . | . | . | . | . | . | . |
| 2009.ICPC.Capobianco [25] | 24 | ✓ | ✓ | ✓ | ✓ | . | . | . | . | . | ✓ | ✓ | . | . | ✓ | . | . | . | ✓ | . | . | . | . | . | . | . |
| 2010.ICPC.Oliveto [26] | 63 | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | ✓ | ✓ | ✓ | ✓ | . | . | . | ✓ | . | ✓ | . | . | . | . | . |
| 2010.ICSE.Asuncion [27] | 76 | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | ✓ | . | ✓ | . | . | . | . | ✓ | . | . | . | . | . | . | . |
| 2011.ICPC.DeLucia [28] | 11 | ✓ | ✓ | ✓ | ✓ | . | . | . | ✓ | . | . | ✓ | ✓ | . | . | . | . | . | ✓ | . | . | . | . | . | . | . |
| 2011.ICSE.Chen [29] | 4 | ✓ | . | . | . | . | . | . | . | . | . | . | ✓ | . | . | . | . | . | ✓ | . | . | . | . | . | . | . |
| 2011.ICSM.Gethers [30] | 24 | ✓ | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | ✓ | . | ✓ | ✓ | . | . | ✓ | . | ✓ | . | ✓ | . | . | . |
| 2013.CSMR.Panichella [31] | NA | ✓ | ✓ | . | . | . | ✓ | . | . | . | . | . | ✓ | . | ✓ | . | . | . | ✓ | . | . | . | . | ✓ | ✓ | . |
| 2013.ICSE.Panichella [32] | 5 | ✓ | . | . | . | . | . | . | . | . | . | . | . | ✓ | . | . | . | . | ✓ | ✓ | . | . | . | . | . | ✓ |
| 2013.TEFSE.Dit [33] | 2 | ✓ | . | . | . | . | . | . | . | . | . | . | . | ✓ | . | . | . | . | ✓ | . | . | . | . | . | . | ✓ |
| **Feature Location** | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2004.WCRE.Marcus [34] | 260 | ✓ | . | . | ✓ | . | . | . | . | . | . | ✓ | . | . | . | . | . | . | ✓ | . | . | . | . | . | . | . |
| 2007.ASE.Liu [35] | 96 | ✓ | ✓ | . | ✓ | X | . | . | . | . | . | ✓ | . | . | . | . | . | . | . | ✓ | . | ✓ | ✓ | . | . | . |
| 2007.TSE.Poshyvanyk [36] | 191 | ✓ | ✓ | . | ✓ | . | . | . | . | . | . | ✓ | . | . | . | . | . | . | . | ✓ | . | . | ✓ | . | . | . |
| 2009.ICPC.Revelle [37] | 33 | ✓ | . | . | . | X | ✓ | . | . | . | . | ✓ | . | . | . | . | . | . | . | . | . | ✓ | . | . | . | . |
| 2009.ICSM.Gay [38] | 39 | ✓ | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | ✓ | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 2011.ICPC.Dit [39] | 23 | ✓ | ✓ | ✓ | ✓ | X | . | X | . | . | . | ✓ | . | . | . | . | . | . | . | ✓ | . | ✓ | . | . | . | . |
| 2011.ICPC.Scanniello [40] | 8 | ✓ | ✓ | ✓ | ✓ | . | ✓ | . | . | . | . | . | ✓ | . | . | . | . | . | . | ✓ | . | . | . | . | . | . |
| 2011.ICSM.Wiese [41] | 4 | ✓ | . | ✓ | . | . | . | . | . | ✓ | . | . | ✓ | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 2012.ICPC.Dit [42] | 9 | ✓ | ✓ | ✓ | ✓ | X | . | . | . | . | . | ✓ | ✓ | . | . | . | . | . | . | ✓ | . | ✓ | . | . | . | . |
| 2013.EMSE.Dit [23] | 6 | ✓ | ✓ | ✓ | ✓ | X | ✓ | . | . | . | . | ✓ | . | . | . | . | ✓ | ✓ | . | ✓ | . | ✓ | . | . | . | . |
| **Program Comprehension** | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2009.MSR.Enslen [43] | 57 | ✓ | . | . | ✓ | . | . | X | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 2009.MSR.Tian [44] | 34 | ✓ | ✓ | . | ✓ | . | . | . | . | . | . | . | . | ✓ | . | . | . | . | ✓ | . | . | . | . | . | . | . |
| 2010.ICSE.Haiduc [45] | 27 | ✓ | ✓ | ✓ | ✓ | . | . | . | . | . | . | ✓ | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| 2012.ICPC.DeLucia [46] | 5 | ✓ | ✓ | ✓ | ✓ | . | . | . | . | . | . | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | . | . | . | . | . |
| **Identify Duplicate Bug Rep.** | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2007.ICSE.Runeson [47] | 161 | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | . | ✓ | . | . | . | . | . | ✓ | . | . | . | . | . | . | . |
| 2008.ICSE.Wang [48] | 169 | ✓ | ✓ | ✓ | . | X | . | . | . | . | . | . | ✓ | . | . | . | . | . | ✓ | . | . | . | . | . | . | . |
| 2012.CSMR.Kaushik [49] | 7 | ✓ | ✓ | ✓ | . | . | . | . | . | . | . | ✓ | ✓ | ✓ | . | . | . | . | ✓ | . | . | . | . | . | . | . |
| Total: | 1,388 | 27 | 20 | 17 | 14 | 6 | 4 | 2 | 1 | 1 | 1 | 14 | 14 | 7 | 5 | 1 | 1 | 1 | 14 | 7 | 2 | 5 | 3 | 1 | 1 | 2 |

such as its name, description, inputs, and outputs. This declaration allows the class to be registered in TraceLab as a component, and ensures that a component can only be connected with a compatible component. 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*.

The structure of the *Component Library* mirrors the *CDK* hierarchy, providing a mapping from TraceLab to the *CDK*. Components can be organized in TraceLab through the use of 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 II). We implemented a few weighting schemes (*e.g.*, binary term frequency, tf-idf, and log-idf) and similarity functions (*e.g.*, cosine, Jaccard), which 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.

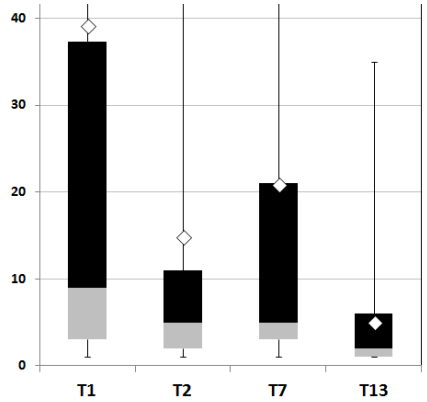


Figure 5 Box plots of effectiveness measure obtained from reproducing the experiments in [23]. The results of techniques $T_1$, $T_2$, $T_7$ and $T_{13}$ correspond to $IR_{LSI}Dyn_{bin}$, $IR_{LSI}Dyn_{bin}WM_{PR(freq)}^{t80}$, $IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}^{b60}$, $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}^{b80}$ from [23] Figure 4(c)

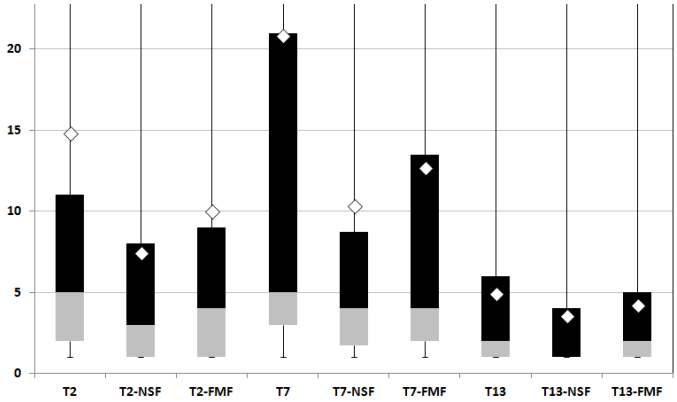


Figure 6 Box plots of effectiveness measure comparing (i) the techniques that produced the best results in [23] for the jEdit4.3 dataset (*e.g.*, $T_2$, $T_7$, and $T_{13}$ - see Figure 5 for exact names) against the (ii) **N**o **S**cenario **F**ilter (suffix NSF) and (iii) **F**requent **M**ethods **F**ilter (FMF)

### C. 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[2].

### D. 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.

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 create (custom / user made) components through the TraceLab SDK. 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 investigating effective ways of incorporating user-made components into the *CL* and *CDK*.

## VI. Reproducing Existing Experiments and Evaluating New Ideas using the Component Library

This section presents the details of reproducing an existing feature location technique (FLT) [23] 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. In addition, we illustrate the process of experimenting with two new ideas that are based on the reproduced technique.

### A. Reproducing a Feature Location Technique

The FLT introduced by Dit *et al.* [23], called $IR_{LSI}Dyn_{bin}WM$ (or *IRDynWM* for short), was reproduced in TraceLab using a subset of components from the proposed *CL*. 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 [35]. The SITIR 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 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 [50] and HITS [51], 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 [50]. 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) [51]. 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 [35].

The reproduced *IRDynWM* FLT, where *WM = {PageRank* or *HITS_Aut* or *HITS_Hub},* is presented as a TraceLab experiment in Figure 4 (see components in the upper part of the figure, which are not highlighted by the red dashed line).

The experiment uses the *loop structure* introduced in the latest version of TraceLab to iterate though 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

[2] http://coest.org/coest-projects/projects/semeru/wiki

Table III Descriptive statistics for the box plots presented in Figure 6. 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 |

(*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 *IR Dyn* 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. It is important to note that the components associated with the *IRDynWM* FLT can be configured with user defined thresholds for the percentage of methods to filter [23].

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 LSI, PageRank and HITS algorithms, as well as other scripts to compute the results. Figure 5 shows a subset of the results produced by our TraceLab implementation, which are the same as the ones that were reported in [23] Figure 4(c) for the jEdit dataset. Figure 5 represents the box plots of the effectiveness measure 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 [23] Figure 4(c), using the notation from [23]. 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 example in Figure 5 because they produced the best results for the jEdit dataset [23] and to illustrate that the implemented technique produces the same results as the original technique.

### B. Experimenting with new Ideas

Using the *IRDynWM* FLT [23] 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.

#### 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 [23, 52] which contains 150 execution traces generated while exercising particular scenarios. Based on 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 Figure 4). 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 $IRDynPageRank_{FMF}$ (see section VI.B.3) for results).

#### 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 Figure 4, but for brevity, the diagram is not included in this paper. 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.

*3) Results of the New Ideas*

Figure 6 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 [23], which are $T_2$, $T_7$ and $T_{13}$ (see Figure 5 for the labels). A complementary view of Figure 6 is given by Table III, which contains descriptive statistics of the box plots generated by those techniques.

Figure 6 and Table III 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 5, whereas for $T_2$-FMF and $T_2$-NSF the median values for 4 and 3 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 [23] for the jEdit dataset. However, the improvement in "precision", comes at the cost of potentially filtering out relevant methods. For example in Table III 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.

### C. Discussion

Although for this particular dataset the two experimental ideas produced better results than the ones reported in [23], 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 [53, 54], etc.) before considering these ideas as viable techniques, but 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 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.

## VII. Limitations

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

TraceLab was not designed for real-time feedback from the user, and although this could be implemented it would impose some overhead upon the developer. Additionally, running code hosted in a .NET process is slower than running it natively. Therefore the time or speed factors in evaluating an approach would need to be considered.

We attempted to identify papers which 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. We are continuously working on driving new research with TraceLab 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 release under GPL as well.

## VIII. Conclusions and Future Work

This paper 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 this 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). In addition, we encourage other researchers to contribute to this body of knowledge for the benefit of conducting research.

## Acknowledgment

This work is supported in part by the United States NSF CNS-0959924, CCF-1218129, and 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: Jane Cleland-Huang, Ed Keenan, Adam Czauderna, and Greg Leach. This work would not have been possible without their continuous support on the TraceLab project.

## References

[1] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *Journal of Software: Evolution and Process (JSEP),* vol. 25, pp. 53–95, 2013.

[2] G. Robles, "Replicating MSR: A Study of the Potential Replicability of Papers Published in the Mining Software Repositories Proceedings," in *MSR*, 2010, pp. 171-180.
[3] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney, "The Effect of Omitted-Variable Bias on the Evaluation of Compiler Optimizations," *IEEE Computer,* vol. 43, pp. 62-67, 2010.
[4] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating Defect Prediction Approaches: a Benchmark and an Extensive Comparison," *Empirical Software Engineering (ESE),* vol. 17, pp. 531-577, 2012.
[5] E. Barr, C. Bird, E. Hyatt, T. Menzies, and G. Robles, "On the Shoulders of Giants," in *FoSER*, 2010, pp. 23-28.
[6] J. M. González-Barahona and G. Robles, "On the Reproducibility of Empirical Software Engineering Studies based on Data Retrieved from Development Repositories," *Empirical Software Engineering (ESE),* vol. 17, pp. 75-89, 2012.
[7] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012 ). *The PROMISE Repository of Empirical Software Engineering Data*. Available: http://promisedata.googlecode.com
[8] S. J. Sayyad and T. J. Menzies. (2005 July 17). *The PROMISE Repository of Software Engineering Databases*. Available: http://promise.site.uottawa.ca/SERepository
[9] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in *PROMISE*, 2007.
[10] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering,* vol. 10, pp. 405-435, 2005
[11] J. Cleland-Huang, A. Czauderna, A. Dekhtyar, G. O., J. Huffman Hayes, E. Keenan, G. Leach, J. Maletic, D. Poshyvanyk, Y. Shin, A. Zisman, G. Antoniol, B. Berenbach, A. Egyed, and P. Maeder, "Grand Challenges, Benchmarks, and TraceLab: Developing Infrastructure for the Software Traceability Research Community," in *TEFSE*, 2011.
[12] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hussein, and D. Hearn, "TraceLab: An Experimental Workbench for Equipping Researchers to Innovate, Synthesize, and Comparatively Evaluate Traceability Solutions," in *ICSE*, 2012, pp. 1375-1378.
[13] J. Cleland-Huang, Y. Shin, E. Keenan, A. Czauderna, G. Leach, E. Moritz, M. Gethers, D. Poshyvanyk, J. H. Hayes, and W. Li, "Toward Actionable, Broadly Accessible Contests in Software Engineering," in *ICSE*, 2012, pp. 1329-1332.
[14] Rapid-I. *Rapid Miner*. Available: http://rapid-i.com/content/view/181/190/
[15] Mathworks. *Simulink*. http://www.mathworks.com/products/simulink/
[16] U. of California. *The Kepler Project*. Available: https://kepler-project.org/
[17] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Communications of the ACM (CACM),* vol. 18, pp. 613-620, 1975.
[18] T. U. of Waikato.. *WEKA*. http://www.cs.waikato.ac.nz/ml/weka/
[19] Yahoo. *Yahoo Pipes*. Available: http://pipes.yahoo.com/pipes/
[20] T. U. of Sheffield. *GATE* Available: http://gate.ac.uk/
[21] B. A. Kitchenham, D. Budgen, and O. P. Brereton, "Using Mapping Studies as the Basis for Further Research - A Participant-Observer Case Study," *Information and Software Technology,* vol. 53, pp. 638-651, 2011.
[22] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic Mapping Studies in Software Engineering," in 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08), 2008.
[23] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software," *Empirical Software Engineering,* vol. 18, pp. 277-309, 2013.
[24] A. Abadi, M. Nisenson, and Y. Simionovici, "A Traceability Technique for Specifications," in *ICPC*, 2008, pp. 103-112.
[25] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella, "On the role of the nouns in IR-based traceability recovery," in *ICPC*, 2009, pp. 148-157.
[26] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery," in *ICPC*, 2010, pp. 68-71.
[27] H. Asuncion, A. Asuncion, and R. Taylor, "Software Traceability with Topic Modeling," in *ICSE*, 2010, pp. 95-104.
[28] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Improving IR-based Traceability Recovery Using Smoothing Filters," in *ICPC*, 2011, pp. 21-30.
[29] X. Chen, J. Hosking, and J. Grundy, "A Combination Approach for Enhancing Automated Traceability", in *ICSE*, 2011, pp. 912-915.
[30] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Link Recovery," in *ICSM*, 2011, pp. 133-142.
[31] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Using Structural Information and User Feedback to Improve IR-based Traceability Recovery," in *CSMR*, 2013, pp. 199-208.
[32] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to Effectively Use Topic Models for Software Engineering Tasks? An Approach based on Genetic Algorithms," in *ICSE*, 2013, pp. 522-531.
[33] B. Dit, A. Panichella, E. Moritz, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "Configuring Topic Models for Software Engineering Tasks in TraceLab," in *TEFSE*, 2013, pp. 105-109.
[34] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *WCRE*, 2004, pp. 214-223.
[35] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace," in *ASE*, 2007, pp. 234-243.
[36] D. Poshyvanyk, Y. G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval," *IEEE Transactions on Software Engineering (TSE),* vol. 33, pp. 420-432, 2007.
[37] M. Revelle and D. Poshyvanyk, "An Exploratory Study on Assessing Feature Location Techniques," in *ICPC*, 2009, pp. 218-222.
[38] G. Gay, S. Haiduc, M. Marcus, and T. Menzies, "On the Use of Relevance Feedback in IR-Based Concept Location," in *ICSM*, 2009, pp. 351-360.
[39] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can Better Identifier Splitting Techniques Help Feature Location?," in *ICPC*, 2011, pp. 11-20.
[40] G. Scanniello and A. Marcus, "Clustering Support for Static Concept Location in Source Code," in *ICPC*, 2011, pp. 1-10.
[41] A. Wiese, V. Ho, and E. Hill, "A Comparison of Stemmers on Source Code Identifiers for Software Search," in *ICSM*, 2011, pp. 496-499.
[42] B. Dit, E. Moritz, and D. Poshyvanyk, "A TraceLab-based Solution for Creating, Conducting, and Sharing Feature Location Experiments," in *ICPC*, 2012, pp. 203-208.
[43] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining Source Code to Automatically Split Identifiers for Software Analysis," in *MSR*, 2009, pp. 71-80.
[44] K. Tian, M. Revelle, and D. Poshyvanyk, "Using Latent Dirichlet Allocation for Automatic Categorization of Software," in *MSR*, 2009, pp. 163-166.
[45] S. Haiduc, J. Aponte, and A. Marcus, "Supporting Program Comprehension with Source Code Summarization," in *ICSE*, 2010, pp. 223-226.
[46] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR Methods for Labeling Source Code Artifacts: Is it Worthwhile?," in *ICPC*, 2012, pp. 193-202.
[47] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *ICSE*, 2007, pp. 499-510.
[48] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information," in *ICSE*, 2008, pp. 461-470.
[49] N. Kaushik and L. Tahvildari, "A Comparative Study of the Performance of IR Models on Duplicate Bug Detection," in *CSMR*, 2012, pp. 159-168.
[50] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," in *7th Int. Conference on World Wide Web*, 1998, pp. 107-117.
[51] J. M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment," *Journal of the ACM,* vol. 46, pp. 604-632, 1999.
[52] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. Kagdi, "Generating Benchmarks from Change History Data to Support Evaluation of Software Maintenance Tasks," in *MSR Data Track*, 2013, pp. 131-134.
[53] A. Egyed, "A Scenario-Driven Approach to Trace Dependency Analysis," *Transactions on Software Engineering (TSE),* vol. 29, pp. 116 - 132, 2003.
[54] T. Eisenbarth, R. Koschke, and D. Simon, "Feature-Driven Program Understanding Using Concept Analysis of Execution Traces," presented at the IWPC, 2001, pp. 300-309.