ForkSim: Generating Software Forks for Evaluating Cross-Project Similarity Analysis Tools

Jeffrey Svajlenko Chanchal K. Roy
University of Saskatchewan, Canada
{jeff.svajlenko, chanchal.roy}@usask.ca

Slawomir Duszynski
Fraunhofer IESE, Kaiserslautern, Germany
slawomir.duszynski@iese.fraunhofer.de

***Abstract*—Software project forking, that is copying an existing project and developing a new independent project from the copy, occurs frequently in software development. Analysing the code similarities between such software projects is useful as developers can use similarity information to merge the forked systems or migrate them towards a reuse approach. Several techniques for detecting cross-project similarities have been proposed. However, no good benchmark to measure their performance is available. We developed ForkSim, a tool for generating datasets of synthetic software forks with known similarities and differences. This allows the performance of cross-project similarity tools to be measured in terms of recall and precision by comparing their output to the known properties of the generated dataset. These datasets can also be used in controlled experiments to evaluate further aspects of the tools, such as usability or visualization concepts. As a demonstration of our tool, we evaluated the performance of the clone detector NiCad for similarity detection across software forks, which showed the high potential of ForkSim.**

I. Introduction

In software development, similar software projects called software variants can emerge in various ways including cloning [1]. Although systematic reuse approaches such as software product lines are known to enable considerable effort savings [2], existing projects are frequently just forked and modified to meet the needs of particular clients and users [3]. These variants typically undergo further parallel development and evolution, and reuse techniques are often not explored until after the variants have matured. This leads to an increased maintenance effort as many tasks are duplicated across the variants.

Maintenance effort can be reduced by merging the forks or by adopting a software reuse approach (e.g., software product lines). Berger et al. [4] report that 50% of industrial software product lines developed by participants of their study were created in an extractive way, i.e., by merging already existing products. This indicates a substantial practical demand for cross-project similarity detection approaches that help software developers discover the similarities between their software variants and support decisions on reuse adoption strategy. Several such approaches have been proposed (e.g., [5] [6]).

A current need is a benchmark for evaluating the performance of tools which detect similarity between software variants. Performance is measured in terms of recall and precision. Recall is the ratio of the similar source code shared between the variants that the tool is able to detect and report. Precision is the ratio of the similar source code elements reported by the tool which are not false positives.

Measuring recall and precision involves executing the tool for a benchmark dataset (or datasets) and analysing the tool's output. Precision can be easily measured by manually validating all (or typically a random subset) of the tool's output. However, measuring recall requires that all similar code amongst the variants of the dataset be foreknown. This makes it very difficult to use datasets from industry or open source (e.g., BSD Unix forks). Building an oracle by manually investigating the dataset for similar code is, due to time required, essentially impossible for datasets large enough to allow meaningful performance evaluation.

To address these difficulties, and to reduce the amount of required manual validation to a minimum, we developed ForkSim, which uses source code mutation and injection to construct datasets of synthetic software forks. The forks are generated such that their similarities and differences at the source level are known. Recall can then be measured automatically by procedurally comparing a tool's output against the dataset's known similarities. Precision can be measured semi-automatically, as reported similarities which match known similarities or differences in the dataset can be automatically judged as true or false positives, respectively. Only the reported similar code not matching known properties needs to be manually investigated.

The forks generated by ForkSim can be used in any research on detecting, visualizing, or understanding code similarity among software products. The generated forks are a good basis for evaluating automated analysis approaches, as well as for performing controlled experiments to investigate how well the specific tool supports users in understanding similar code. ForkSim is publicly available for download[1].

This paper is organized as follows. Related work is discussed in Section II, and software forking in Section III. Section IV outlines ForkSim's fork generation process, Section V outlines the comprehensiveness of the simulation, and Section VI discusses the quality of the generated forks. Section VII outlines ForkSim's use cases, while Section VIII provides a demonstration of its primary use case: tool performance evaluation. Section IX discusses our plans for future experiments with ForkSim, and Section X our future work on ForkSim. Finally, Section XI concludes the paper.

978-1-4673-5739-5/13/$31.00

[1] http://homepage.usask.ca/~jes518/forksim.html

II. Related and Previous Work

Although automatic tool benchmark construction has been proposed in various problem domains [7], to the best of our knowledge there are no other tools which generate software fork datasets for evaluating cross-project similarity analysis tools, nor is there a reference case (e.g., a set of software forks with known properties) which could be used for tool evaluation. The most related work to ours is a manual validation of cross-project similarity analysis results obtained through clone detection by Mende et al. [6]. However, manual result validation has several drawbacks, as discussed in the introduction. ForkSim is unique in that tool evaluation can be mostly automated for datasets generated by ForkSim, as the similarities and differences between the generated software forks are known.

In previous work, we built a framework which automatically evaluates clone detection tools for intra-project clones by generating a dataset of artificial clones using source code fragment mutation and injection [8] [9] [10]. In this work, we use source mutation and injection at a number of source granularities (function, file, directory) to simulate a forking scenario. The result is a set of software variants with known similarities and differences which can be used to measure the performance of cross-project similarity detection tools.

III. Software Forking

In the forking process, a software system is cloned and the individual forks are evolved in parallel. Development activities may be unique to an individual fork, or shared amongst multiple forks. For example, code may be copied from one fork to another. While forks may share source code, the code may contain fork-specific modifications, and may be positioned differently within the individual forks. A fork may itself be forked into additional forks. Table I provides a taxonomy of the types of source code level development activities performed on forks. These development activities describe how a fork may change with respect to its state at the start of the forking process. This taxonomy is based upon our research and development experience, and discussions with software developers.

Existing code originates from pre-fork development, and new code originates from post-fork development. The development activities occur at various code granularities, including: source directory, source file, function, etc. The result of forking and these further development activities are a set of software variants containing source code in the following three categories: (1) code shared amongst all the forks, (2) code shared amongst a proper subset of the forks, and (3) code unique to a specific fork. ForkSim creates datasets of forks resulting from these development activities, and containing source code in these three categories. It does this by simulating a simple forking scenario.

TABLE I
Taxonomy of Fork Development Activities

ID	Development Activity
DA1	New source code is added.
DA2	Existing source code is removed.
DA3	Existing source code is modified and/or evolved.
DA4	Existing source code is moved.
DA5	Source code is copied from another fork. It may be copied into a different position than in the source fork, and it may be modified and/or evolved independently of the source.
DA6	A fork may itself be forked.

IV. Fork Generation

ForkSim's generation process begins with a base subject system, which is duplicated (forked) a specified number of times. Continued development of the individual forks is simulated by repeatedly injecting source code into the forks. Specifically, ForkSim injects a user specified number of functions, source files, and source directories. Instances of source code of these types are mined from a repository of software systems, which ensures the injected code is realistic and varied.

Each of the chosen functions, files and directories are injected into one or more of the forks. The number and particular forks to inject a source artefact into are selected at random. Injection into a single fork creates code unique to that fork, while injection into a subset of the forks creates code shared amongst those forks. Injection locations are selected randomly, but only amongst the code inherited from the base system, i.e., not inside previously injected code. This prevents the injected code from interacting, which makes the generation process much easier to track and thereby simplifies tool evaluation using the generated dataset. When code is injected into multiple forks, the injection location may be kept uniform or varied, given a specified probability. Forks may share code, but that code may be positioned differently within the individual forks.

Before code is injected into a fork it may be mutated, i.e., automatically modified in a defined way, given a specified probability. This causes code shared by the forks to contain differences, simulating that shared code may be modified or evolved independently for the needs of a particular fork.

Files and directories may be renamed before injection, given a specified probability. While forks may share code at the file and directory level, they may not have the same name. For example, they may have been renamed to match conventions used in the fork.

Each injection operation (injection of a function, a file, or a directory) is logged. This includes recording the forks the code was injected into, the injection locations used, and if and how the code was mutated and/or renamed before injection. A copy of the code and any of its mutants are kept separately and referenced by the log. ForkSim also maintains a copy of the original subject system. From the log and the referenced data the directory, file and function code similarities and differences inherited from the original system and introduced by injection, can be procedurally deduced.

Fig. 1 depicts this generation process. On the left side are the inputs: the subject system, source repository and user-specified generation parameters. The subject system duplicates (forks) are modified by the boxed process, which repeats for each of the source files, directories and functions to be introduced. The figure shows an example of this process, in which a randomly selected function from the source repository is introduced into the forks. ForkSim randomly decided to

inject this function into three of the four forks using non-uniform injection locations, and to mutate the function before injection into the latter two forks. On the right side are the outputs: the generated fork dataset and the generation log.

ForkSim supports the generation of Java, C and C# fork datasets. It is implemented in Java 7 (main architecture and simulation logic) and TXL [11] (source code analysis, extraction and mutation). ForkSim's generation parameters are summarized in Table II.

Injection. Directory and file injection is accomplished by copying the directory or file into a randomly selected directory in the fork. In order to prevent injected directories from dominating a fork's directory tree, only leaf directories (those containing no subdirectories) are selected for injection. Function injection is performed by selecting a random source file from the fork, and copying the function's content directly after a randomly selected function in the chosen file. The generation process can be parametrized to only select functions for injection which fall within a specified size range measured in lines before mutation.

Mutation. ForkSim uses mutation operators to mutate code before injection. Fifteen mutation operators, named and described in Table III, were implemented in TXL [11]. Each operator applies a single random modification of its defined type to input code. These mutation operators are based upon a comprehensive taxonomy of the types of edits developers make on cloned code [12], which makes them suitable for simulating how developers modify shared code duplicated between forks.

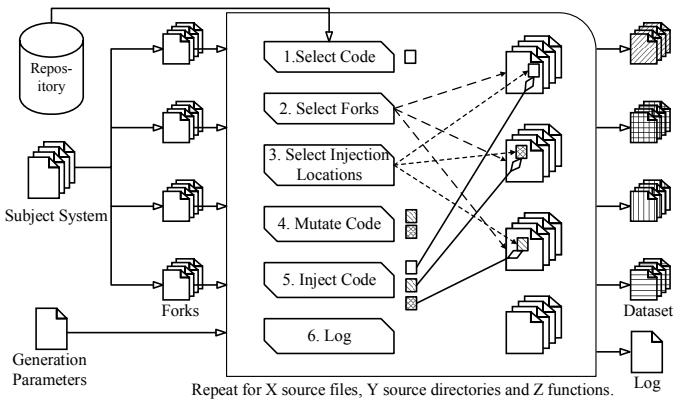

Fig. 1. Fork Generation Process

TABLE II
FORKSIM GENERATION PARAMETERS

Parameter	Description
Subject System	The base system which is forked during the generation process.
Source Repository	A collection of systems from which the source files, directories and functions are mined.
Language	Language of forks to generate (Java, C or C#).
# Forks	Number of forks to generate.
# Files	Number of files to inject.
# Directories	Number of directories to inject.
# Functions	Number of functions to inject.
Function Size	Maximum/Minimum size of functions to inject.
Max Injections	Maximum number of forks to inject a particular function/file/directory into.
Uniform Injection Rate	Probability of uniform injection of a source artefact.
Mutation Rate	Probability of source mutation before injection. Specified separately for function, file and directory injections.
Rename Rate	Probability of renaming before injection.
Max Mutations	Maximum number of mutations to apply to injected code. Specified as a ratio of the code's size in lines.

Files and functions are mutated by applying one of the mutation operators a random number of times before injection. The number of mutations is limited to a specified ratio of the size of the file/function measured in lines after pretty printing. This provides an upper limit on how much simulated development is allowed to occur on a source file or function in a particular fork. Pretty printing (one statement per line, no empty lines, comments removed) the source artefact before measurement ensures the measure is consistently proportional to the amount of actual source code contained. This ratio can be specified separately for files and functions.

A small mutation ratio is recommended (10-15%) as too many changes may cause the variants of a file/function injected into multiple forks to become so dissimilar that they would no longer be a clone. Detection tools would be correct not to report them. ForkSim datasets are only useful if the elements declared as similar are indeed similar.

When directories are injected, each of the source files in the directory may be mutated using the same process as used for injected files. The directory mutation probability parameter defines how likely a file in an injected directory is mutated.

As a principle of mutation analysis, ForkSim does not mix mutation operators. This makes it easier to discover if a similarity analysis tool struggles to detect similar code with particular types of differences. ForkSim cycles through the mutation operators to ensure that each is represented in the generated forks roughly evenly. When it is not possible to apply a given operator to the file or function, another operator is chosen randomly.

Renaming. The probability of a file or directory being

TABLE III
MUTATION OPERATORS FROM A CODE EDITING TAXONOMY FOR CLONING

ID	Description
mCW_A	Change in whitespace (addition).
mCW_R	Change in whitespace (removal).
mCC_BT	Change in between token (/* */) comments.
mCC_EOL	Change in end of line (//) comments.
mCF_A	Change in formatting (addition of a newline).
mCF_R	Change in formatting (removal of a newline).
mSRI	Systematic renaming of an identifier.
mARI	Arbitrary renaming of a single identifier.
mRL_N	Change in value of a single numeric literal.
mRL_S	Change in value of a single string literal.
mSIL	Small insertion within a line (function parameter).
mSDL	Small deletion within a line (function parameter).
mIL	Insertion of a line.
mDL	Deletion of a line.
mML	Modification of a whole line.

renamed before injection is specified separately from that of source code mutation, and both are allowed to occur on the same injection. Renamed source files keep their original extensions.

Usage. ForkSim operation is very simple. The user makes a copy of the default generation parameters file, and tweaks it for the dataset they wish to generate. This includes specifying paths to the subject system and source repository to use. Once the parameters file and systems are prepared, the user executes ForkSim and specifies the location of the parameters file and a directory to output the forks and generation log into. Once execution is complete, the forks and generation log are ready for use in experiments involving similarity analysis tools.

V. Simulation of Development Activities

During the generation process, ForkSim simulates all six of the development activities from the forking taxonomy (Table I). The following subsections describe how the code injection scenarios performed during the generation process can be interpreted as the six development activities.

DA1. Any of the code injections can be interpreted as the addition of new code to a fork.

DA2. Code injected into a proper subset of the forks can be interpreted as existing code (pre-fork) which was deleted from the forks it was not injected into.

DA3. Code injected into the forks, with at least one instance mutated, can be interpreted as existing code which was modified/evolved in one or more of the forks, perhaps inconsistently. The code needs not be injected into all of the forks to simulate *DA3*, as the forks missing the code can be interpreted as having lost this shared code due to *DA2*.

DA4. Code injected into the forks, with variation in injection location, can be interpreted as existing code being moved. When the code is not injected into all the forks, the forks missing this existing code can be interpreted as instances of *DA2*.

DA5. Code injected into multiple forks can be interpreted as code implemented in one fork and copied into others. Non-uniform injection, source mutation and renaming simulate that the code may be copied into a different location than in the source, and continued development may occur independently of the source.

DA6. While the generation process creates all of the forks from the same forking point (the base system), the resulting dataset can be interpreted as originating from multiple forking points. Code shared due to injection amongst a subset of the variants can be interpreted as development before a shared forking point, which may not be shared across all the forks. This activity can also be simulated by using the forks as the base system for additional executions of ForkSim.

VI. Discussion

Advantages. The primary advantage of ForkSim is that the user can precisely control the amount and type of similarities and differences among the generated forks. This allows for well-controlled evaluation of tools which analyse forks. Moreover, as the fork generation process is known and logged, the correct and complete information on the actual code similarities and differences between the forks is available. This is not the case when real-world forks are analysed.

Disadvantages. One of the limitations of ForkSim is that the generated variants may have properties that differ from real forks, particularity if aggressive injection settings are used. As injection is a random process, the code-level properties do not represent meaningful development. Also, the distribution of the similar and dissimilar code might differ from real-world forks. However, to the best of our knowledge, there are no systematic studies on the amount and distribution of code similarities and differences in real-world forks. Therefore we are not able to tune our generation algorithm and its parameters to produce very realistic forks. However, as fork analysis tools likely do not behave differently for less realistic software variants, it is unlikely that this will have a significant effect on tool evaluations using ForkSim-generated fork datasets.

Unknown Similarities. The similarities and differences between the forks inherited from the subject system and intentionally created by injection are exactly known. However, there will be some additional similarities between the forks which are unknown. These include: (1) clones within the original subject system which become similar code within and between the generated forks, (2) unexpected similarity between the functions, files, and directories randomly chosen from the source repository, and (3) unexpected similarity between these chosen source artefacts and the subject system.

Since these similarities are unknown, they are not included in the measurement of a tool's recall for the dataset. However, this is not a disadvantage as we are not interested in evaluating the tools for intra-project similarities. The known similarities are sufficient for measuring cross-project similarities. These similarities, however, must be considered in the measure of a tool's precision.

VII. Use Cases

Cross-Project Similarity Tool Performance Evaluation. ForkSim datasets can be used to measure the recall and precision of tools which detect similarity between software projects. It is especially attuned for tools which focus on similarity detection between software variants (e.g. forks). ForkSim datasets are ideal for this usage scenario as similarities and differences between the generated forks are known. Recall can then be measured automatically, and precision semi-automatically. Recall is evaluated by measuring the ratio of the similar code between the forks, and their relationships, the tool is able to detect and report. The recall measure considers both the similarities created by injection, and the similarities between each of the forks due to the duplication of the base system during the generation process.

How tools report similar code is likely to differ. Therefore, to evaluate recall the dataset's generation log must be mined and converted into the detection tool's output format. This process creates the tool's gold standard, i.e., its ideal and perfect output for the dataset. Recall is then the ratio of the gold standard the tool was able to produce. By building the gold standard procedurally, recall evaluation becomes automatic. The conversion procedure needs only to be written once, and reused for various datasets.

Precision can be evaluated semi-automatically. Any detected similar code which is in the gold standard can be automatically labelled as true positive. Any reported similarities which match known differences in the dataset can be labelled as false positives. The remaining output requires manual validation to complete the measure of precision. It is sufficient to validate a random subset (large enough to be statistically significant) of the remaining output and to estimate precision from these results.

Tool Usability Study. ForkSim-generated datasets are valuable for performing controlled experiments involving tools which analyse and/or visualize similarities and differences between software variants. The goal of such an experiment can be to measure the level of support for software similarity comprehension the tools provide to their users. The experiment would have the following procedure: first, a dataset of forks with known similarities is generated using ForkSim. Then, the study participants, divided into a few groups, use the tools to analyse the dataset and report their findings. Each user group uses a different tool to solve the same tasks. For example, the participants can be asked a set of questions related to the similarity of the analysed variants, which they should answer by discovering and understanding the similarities using the given tool. The tasks should be designed to evaluate a specific aspect of the tools, e.g. their usability or the appropriateness of the used similarity visualizations. By checking the correctness of the answers and recording the amount of needed effort and/or time, user group performance is quantitatively measured. In this way, the effect of using the different tools is quantified, and the tools can be compared regarding the properties targeted by the tasks, such as tool usability. As discussed in Section VI, ForkSim-generated datasets have both advantages (e.g. precise control of similarity) and disadvantages (e.g. injection of non-related code) as compared to real datasets. Hence, the use of generated datasets is suitable in situations where the stated disadvantages do not influence the experiment goal.

Adaptations. For the purpose of clone management, detecting and studying clone genealogies is another important research topic, and there have been a few genealogy detectors (e.g., [13]). The technology used in ForkSim can easily be adapted to generate software versions rather than software variants for evaluating genealogy detector performance. Such an adaptation could also be used to evaluate clone ranking algorithms [14], which use multiple versions of a software system to produce a clone ranking.

VIII. Evaluation

As a demonstration of ForkSim's primary use case, tool evaluation, we evaluated NiCad's performance for similarity detection between software variants. NiCad [15], [16] is one of the state of the art near-miss software clone [1] detectors. While it is designed for single systems, it can be used to detect similarity between forks by executing it for the entire dataset and trimming the intra-project clone results from its output. To evaluate NiCad, we generated a ForkSim dataset of five Java forks. We used JHotDraw54b1 as the subject system and Java6 as the source repository. The generation parameters used are listed in Table IV. The NiCad clone detector is capable of detecting function and block granularity near-miss clones. It uses TXL to parse source elements of these granularities from an input system, and uses a diff-like algorithm to detect clones after these source elements have been normalized to remove irrelevant differences (e.g., formatting, comments, whitespace, identifier names, and more). For use in this experiment, we extended NiCad to support the detection of clones at the file granularity.

Using NiCad, we detected the file and function clones in the dataset. NiCad was set to detect clones 3-5000 lines long, with at most 30% difference. It was configured to pretty print the source, blind rename the identifiers, and normalize the literal values in the dataset before detection. The clones were collected both in clone pair (pairs of similar files or functions) and clone class (set of similar files or functions) format. Overall, NiCad found 363 file clone classes (16,553 pairs) and 1831 function clone classes (2,198,636 pairs).

To evaluate NiCad's recall, we converted the known similarities between the forks into file and function clone classes. Each file injected into multiple forks was converted into a file clone class, as were the files contained in directories injected into multiple forks. File clone classes were created for each of the files the forks inherited from the subject system, with the files modified due to function injection trimmed from these classes. Each function injected into multiple forks was converted into a function clone class. Lastly, a function clone class was created for each function the forks inherited from the subject system. These clone classes were also converted to clone pair format.

NiCad's recall performance is summarized in Table V. Recall was measured per clone granularity (file or function), and per origin of similarity (file/directory/function injection or original subject system files). As can be seen, NiCad had 100% recall for all sources of file clone classes, and 98-99% for function clone classes. If we consider clone pairs instead of clone classes, we see that the function clone detection is marginally better (+0.1%). These are very promising results for NiCad as a fork similarity analysis tool. These results are specific to the dataset's generation parameters. In future we plan to evaluate NiCad's recall performance for many datasets with varied parameters; for example, with larger and smaller max mutation values.

Due to time constraints, we did not perform a full precision

TABLE IV
ForkSim Generation Parameters: NiCad Case Study

Parameter	Value
Subject System	JHotDraw54b1
Source Repository	Java6
Language	Java
# Forks	5
# Files	100
# Directories	25
# Functions	100
Function Size	20-100 lines
Max Injections	5
Uniform Injection Rate	50%
Mutation Rate	files: 50%, directories(files): 50%, functions: 50%
Rename Rate	files: 50%, directories: 50%
Max Mutations	15% of size in lines

TABLE V
NICAD CASE STUDY RECALL RESULTS

Type	File Injections	Directory Injections	Function Injections	Original Files
File Clone Class	100% (41/41)	100% (117/117)	-	100% (260/260)
Function Clone Class	-	-	98.7% (75/76)	99.4% (2869/2886)
Function Clone Pair	-	-	98.8% (332/336)	99.5% (28708/28860)

analysis for this experiment. However, NiCad is known to have high precision [16]. Using known similarities, we were able to validate 20.7% of NiCad's reported file clone pairs, but only 1.46% of its reported function clone pairs. NiCad is reporting a large amount of cloned code beyond that of the known similarities. Part of this is due to unknown similarities arising from clones within the original subject system. However, a large fraction of this is due to the NiCad clone size settings used. A minimum clone size of 3 lines was required to ensure that all cloned functions were detected. However, small standard functions such as getters and setters are very similar after normalization, which was a source of a large number of these clone pairs. Likewise, interfaces and simple classes are likely to be detected as similar after identifier normalization. For practical usage, these small similarities would likely be filtered out in preference of the larger similarities. In summary, NiCad has very good detection performance of similarities between forks, but the quantity of output would make its usage difficult. A post-processing step needs to be added to extract the most useful and important similarity features from its output.

IX. FUTURE EXPERIMENTS

We plan to use ForkSim to evaluate our tools for similarity detection between software variants. We will use a variety ForkSim datasets to evaluate the Fraunhofer Variant Analysis [5] tool's recall and precision. This tool locates and visualizes similarities between software variants with the aim of supporting software product line adoption. We also plan to perform a more in-depth analysis of NiCad's, SimCad's [17], another state of the art near-miss clone detector, and other clone detectors' performance in this domain. These experiments are prime examples of ForkSim's featured use cases, and demonstrate how other researchers and practitioners might utilize ForkSim.

X. FUTURE WORK

The following are additional features we plan to add in future work: **(1)** The option to allow the mixing of mutation operators during file and function mutation. **(2)** Expand directory injection beyond leaf directories, with optional constraints on hierarchy size (breadth and depth) of the injected directory. **(3)** The splitting of files or directories before injection, given a defined probability. **(4)** An alternative mode, in which ForkSim injects into a set of disparate systems rather than duplicating (forking) a single system. This would allow the generation of systems with shared code which were not the result of forking, but of large scale code re-use by copy and paste. **(5)** Using tools we have tested and evaluated with ForkSim datasets, we plan to study the similarity patterns found in real forked systems. We will use our findings to enhance ForkSim and tune its parameters to produce datasets which better mimic real-world forks.

XI. CONCLUSION

In this paper we have introduced ForkSim, a tool for generating customizable datasets of synthetic forks with known similarities and differences. These datasets can be used in any research on the detection, visualization, and comprehension of code similarity amongst software variants. ForkSim datasets allow similarity detection tools to be evaluated in terms of recall (automatically) and precision (semi-automatically), and can be useful in experiments aiming at evaluating the usability and visualization of similarity tools. We demonstrated ForkSim using a case study evaluating NiCad's cross-project similarity detection for a set of five ForkSim-generated Java forks.

REFERENCES

[1] C. K. Roy, "Detection and analysis of near-miss software clones," in *Proc. ICSM*, 2009, pp. 447–450.

[2] P. Clements and L. Northrop, *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[3] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," *Proc. CSMR*, pp. 25–34, 2013.

[4] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *Proc. VaMoS*, 2013, pp. 7:1–7:8.

[5] S. Duszynski, J. Knodel, and M. Becker, "Analyzing the source code of multiple software variants for reuse potential," in *Proc. WCRE*, 2011, pp. 303–307.

[6] T. Mende, R. Koschke, and F. Beckwermert, "An evaluation of code similarity identification for the grow-and-prune model," *Journal of Software Maintenance and Evolution*, vol. 21, no. 2, pp. 143–169, Mar. 2009.

[7] D. Lo and S.-C. Khoo, "Quark: Empirical assessment of automaton-based specification miners," in *Proc. WCRE*, 2006, pp. 51–60.

[8] J. Svajlenko, C. Roy, and J. Cordy, "A mutation analysis based benchmarking framework for clone detectors," in *Proc. IWSC*, May 2013, pp. 8–9.

[9] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Proc. ICSTW*, 2009, pp. 157–166.

[10] ——, "Towards a mutation-based automatic framework for evaluating code clone detection tools," in *Proc. C3S2E*, 2008, pp. 137–140.

[11] J. R. Cordy, "The txl source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, Aug. 2006.

[12] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection, techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, pp. 470–495, 2009.

[13] R. Saha, C. Roy, and K. Schneider, "An automatic framework for extracting and classifying near-miss clone genealogies," in *Proc. ICSM*, 2011, pp. 293–302.

[14] M. F. Zibran and C. K. Roy, "A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring," in *Proc. SCAM*, 2011, pp. 105–114.

[15] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. ICPC*, 2008, pp. 172–181.

[16] J. Cordy and C. Roy, "The nicad clone detector," in *Proc. ICPC*, 2011, pp. 219–220.

[17] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *Proc. WCRE*, 2011, pp. 13–22.