Code Relatives: Detecting Similarly Behaving Software

Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey,
Simha Sethumadhavan, Gail Kaiser and Tony Jebara
Columbia University
500 West 120th St, MC 0401
New York, NY USA
{mikefhsu, jbell, harvey, simha, kaiser, jebara}@cs.columbia.edu

1. INSIGHTFUL

Identifying similar code can support many software engineering tasks such as code search and software refactoring. Many excellent approaches have been proposed to detect code having similar 1. static patterns such as syntaxes and/or tokens (code clones), 2. functional patterns defined by programs' inputs and outputs (simions). However, the code snippets that behave alike in executions at fine granularity may be ignored. We develop DyCLINK to detect these behaviorally similar code at instruction level. We name these behaviorally similar code as *code relatives*.

Detecting behaviorally similar code can be hard for the approaches observing static patterns [7], because they do not execute code to collect behavioral information. Thus, these approaches may ignore code snippets having similar behaviors with dissimilar syntaxes. Jiang and Su proposed to identify functionally similar code by observing inputs and outputs of programs [4], a technique referred to as simions by Deissenboeck et al., who extended the idea to object oriented languages [2]. However, Deissenboeck et al. reported low detection rate caused by difficulties in matching different, complex data types used by different programs.

Our insight is that instead of observing the inputs and outputs of programs, we observe how programs compute their results (i.e., execution traces) at the instruction level. Instead of recording these execution traces in sequences, we encode them in a more informative data structure: dynamic instruction graph, where a node is an instruction and an edge is a dependency between two instructions. If two programs have similar dynamic instruction (sub)graphs, they are code relatives.

2. USEFUL

Code relatives can help developers understand program behaviors and search for behaviorally similar code in their systems. However, detecting code relatives is expensive due to the high complexity of the (sub)graph isomorphism problem. Thus, we developed a link analysis based algorithm to efficiently solve this problem. Our experimental results show that DyCLINK detects some code similarities not detected by either a static analysis system or a simion detector, and vice versa.

DyCLINK can cluster programs with similar behavior. We can then normalize the programs in the same cluster to create a common API to enhance system cohesion. We could also build a code search engine to retrieve programs having behavior befitting the user query, while most of the current approaches search for code based on key words or I/Os of programs. DyCLINK might help onboard a new developer by finding and showing them any similarities between the new codebase and other codebases that this developer has worked with. DyCLINK could integrate with record-and-replay and code animation to visualize code relatives side by side.

3. USABLE

DyCLINK can be downloaded from Github under the open-source MIT license [1]. A detailed tutorial describing how to set up, build and execute the system, and how to analyze the detected code relatives, is provided on the same Github page. To facilitate utilizing DyCLINK for future research, we have prepared a virtual machine that contains our system and all required software. This virtual machine is also accessible from our Github page. We use Apache Maven [5] as the build management tool.

4. REFERENCES

[1] Dyclink github page. https://github.com/Programming-Systems-Lab/dyclink.
[2] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner. Challenges of the dynamic detection of functionally similar code fragments. CSMR '12, 2012.
[3] Oracle jdk 7. http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.
[4] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. ISSTA '09, 2009.
[5] Apache maven. https://maven.apache.org.
[6] Mysql database. https://www.mysql.com.
[7] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7), 2009.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FSE '16 Seattle, WA USA
© 2016 ACM. ISBN XXXXX…$15.00
DOI: XX.XXX/XXX_X

A. TUTORIAL OF DYCLINK

We provide a step-by-step tutorial to replay the result of Table 3 in our research paper. A virtual machine (VM) containing DyCLINK and all required software can be accessed from DyCLINK's Github page [1]. Users can first read §A.7 to check the VM's limitation. For replaying our experiments on real machines, we also provide the executable of DyCLINK (`dyclink_fse.tar.gz`) in the same package of the VM. We conducted our experiments on an iMac with 8 cores and 32 GB memory to construct graphs (§A.4) and Amazon ec2 "c4.8xlarge" instances to match graphs (§A.5).

A.1 Required Software Suites

If the user chooses to use our VM, this step can be skipped. The user needs to install JDK 7 [3] to execute our experiments on DyCLINK. DyCLINK is a Maven project [5]. If the user wants to re-compile DyCLINK, the installation of Maven is required. DyCLINK needs a database system and GUI to store/query the detected code relatives. We use MySQL and MySQL Workbench. For downloading and installing them, the user can check MySQL's website [6]. For setting up the database, the user can find more details in `dycl_home/scripts/db_setup`, where `dycl_home` represents the home directory of DyCLINK.

A.2 Virtual Machine

We set up the credential with "dyclink" as the username and "Qwerty123" as the password for our VM. The home of DyCLINK is `/home/dyclink/dyclink_fse/dyclink`. For starting MySQL, the user can use the command `sudo service mysql start`. The credential for MySQL is "root" as the username and "qwerty" as the password.

A.3 System Configuration

Before using DyCLINK, the user needs to change to the home directory of DyCLINK. The user first uses the command `./scripts/dyclink_setup.sh` to create all required directories for executing DyCLINK. DyCLINK has multiple parameters to specify in the configuration file: `config/mib_config.json`. For reproducing the experimental results, the user can simply use the this configuration file.

A.4 Dynamic Instruction Graph Construction

We put our codebases for the experiments under `codebase/bin`. The user will find 4 directories from "R5P1Y11" to "R5P1Y14". These 4 directories contain all Google Code Jam projects we used in the paper from 2011 to 2014.

Before executing the projects in a single year, the user needs to specify the graph directory for the `graphDir` field in the configuration file. This is to tell DyCLINK where to dump all graphs. For example, the user sets `graphDir` to `graphs/2011` for storing graphs of the projects in 2011. We have created subdirectories for each year under `graphs`.

We prepare a script to automatically execute all projects in a single year: `./scripts/exp_const.sh $yearDir`. For example, the user can execute all projects in 2011 by the command `./scripts/exp_const.sh R5P1Y11`. Most years can be completed between 0.5 to 3 hours on the VM, but 2013 may cost 20+ hours and need more memory.

The `cache` directory records cumulative information for constructing graphs. If users fail any year, they need to first clean the `cache` directory and reset `threadMethodIdxRecord` in the configuration file to be empty, and re-run every year.

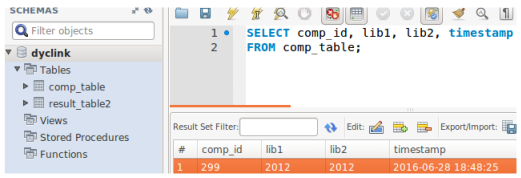
A.5 (Sub)graph similarity computation

Because we compute the similarity between each graph within and between years, there will be totally 10 comparisons. For storing the detected code relatives in the database, the user needs to specify the URL and the username in the configuration file.

For computing similarities between graphs in the same year, the user can issue `./scripts/dyclink_sim.sh -iginit -target graphs/$year`, where `$year` is between $\{2011, 2014\}$. For different years, the user can issue `./scripts/dyclink_sim.sh -iginit -target graphs/$y1 -test graphs/$y2`, where `$y1` and `$y2` are between $\{2011, 2014\}$. DyCLINK will then prompt for user's decision to store the results in the database The user needs to answer "true".

On the VM, we suggest the user to detect code relatives for $2011-2012$, $2011-2014$, $2012-2012$ and $2012-2014$, if we exclude the projects in 2013. The other 6 comparisons may take 20+ hours to complete on the VM.

A.6 Result Analysis



```
SELECT comp_id, lib1, lib2, timestamp
FROM comp_table;
```

#	comp_id	lib1	lib2	timestamp
1	299	2012	2012	2016-06-28 18:48:25

Figure 1: The exemplary UI of MySQL Workbench to check the comparison ID.

For analyzing code relatives for a comparison, the user needs to retrieve the comparison ID from the `dyclink` database. The user first queries all comparisons by the SQL command as Figure 1 shows via MySQL Workbench, and then checks the ID for the comparison. `lib1` and `lib2` show the years (codebases) in a comparison. If the values for `lib1` and `lib2` are different such as $2011-2012$, this comparison contains the code relatives *between* different years. If the values are the same such as $2012-2012$, this comparison is *within* the same year. Figure 1 checks the comparison ID (299) for code relatives within 2012 ($2012-2012$).

For computing the number of code relatives, the user can use the command `./scripts/dyclink_query.sh $compId $insts $sim -f` with 4 parameters. The `$compId` represents the comparison ID. The `$insts` represents the minimum size of code relatives with 45 as the default value. The `$sim` represents the similarity threshold with 0.82 as the default value. The flag `-f` filters out simple utility methods in our codebases. An exemplary command for the $2012-2012$ comparison with `$compId` $= 299$ is `./scripts/dyclink_query.sh 299 45 0.82 -f`.

A.7 Potential Problems

The major potential problem is the performance and memory of VM. Some experiments regarding 2013 may cost too much time and need more memory than the VM has. If the `OutOfMemoryError` occurs, the user can increase the memory for the VM and sets `-Xmx` for JVM in the corresponding commands under the `scripts` directory. For completing *all* experiments in our paper, we suggest to run DyCLINK on a real machine. Also, due to nondeterminism in a running program, DyCLINK may record different graphs, causing results to vary slightly between multiple runs.