CS 621
Idea Proposal Assignment

Due: **Monday, March 20, 2023, 9:00 AM EDT**

Research idea write-up and presentation

This assignment can be done individually or in groups of 2 students. Your choice.

The assignment consists of:

1. Coming up with a creative new research idea.
2. An up to 1-page write-up describing your research idea.
3. A 5-minute presentation, given in class on Thursday, March 23, 2023.

Overview

Your primary job in this assignment is twofold:

1. To describe your proposed research goal so that people understand what it is and why it is valuable. This must include a *research question* you will try to answer.
2. To describe how you will accomplish your research goal and how you will evaluate it so that it is clear how a team of up to three students can answer the research question in approximately 6–7 weeks.

You will present your idea to the class. Everyone will then have the opportunity to review the presentations and form groups of up to three students to actually explore the research idea!

One of the purposes of identifying the research idea is to find an area of software engineering research that is interesting to you. The idea will evolve over time, especially as you read the related work. While this initial idea may differ significantly from the final research question you tackle, the initial idea will serve an important role in focusing you on a particular area of software engineering.

Guidelines and examples

The **research idea** written description **and** the presentation must **each** contain:

- Research question. This must be in the form of a question and describe what you will know after this project is finished that the world does not know today. Examples of reasonable research questions include: "RQ1: Can mutations used in mutation testing generate the kinds of bugs observed in real, open-source development?" and "RQ2: Can the k-tail algorithm for model inference be augmented with information about pre- and post-conditions (inferred by dynamic analysis) to improve the inferred models' precision and recall?"
- The key idea behind the new technique you will develop. For example, for RQ2, the idea may be "Preventing k-tail from merging states if the pre- and post-conditions do not match."
- A concrete evaluation plan that you will use to determine when answering your research question is a success. For example, for RQ1, the plan may be "We will survey the mutation testing literature to enumerate at least 10 top mutation operators. We will then find, on `github.com`, at least 6 open-source programs. Each program will have at least 10K lines of code, 100 actively maintained tests, and at least 1000 commits in its history. We will analyze the commit history to find regression bugs (times

when a test that previously was passing but then started failing), isolate the changes that resulted in each of these bugs, and determine whether these changes could be been generated automatically using the mutation operators we identified."

Deliverables

Using the **Project Idea Assignment** on Gradescope, submit:

- An up to 1-page description of the research idea, in a .pdf. Don't forget to put your name on it.
- A digital presentation (keynote, powerpoint, or pdf). Don't forget to put your name on it.

You will also deliver an in-class presentation. The delivery should take a maximum of 5 minutes. You will be cut off after 5 minutes! Shorter can be OK. Prepare and rehearse your presentation.

You may use any resource you wish in this assignment but you must list your collaborators and cite all your sources. Failure to do so will result in a grade of 0.

Sample project ideas

These ideas are meant as starting points. Students are encouraged to generate their own ideas, or to start from these high-level topics and generate a more concrete idea as a course project.

Generating tests from natural language descriptions.

Software projects are more than just code. There are tests, requirements documents, bug reports, etc. Many of these documents are written in natural language. At the same time, projects' test suites are often incomplete and fail to capture some notions described in these natural-language documents. This project focuses on the problem of generating executable tests from natural-language descriptions of code.

See "C2S: Translating natural language comments to formal program specifications" (https://doi.org/10.1145/3368089.3409716) as a starting point.

One could consider applying recent large-language-model-based code generation tools (see CoPilot, Codex, GPT-J, GPT-Neo, GPT-NeoX-20B, and CodeParrot; https://arxiv.org/abs/2202.13169) for test generation.

Once one has tests, they could evaluate if the tests improve the quality of the existing test suite via coverage or mutation testing.

Model inference algorithm comparison.

There are many model inference tools out there that use logs to produce a finite state machine model (or something close), e.g., Synoptic, InvariMint, kTails, BEAR, Perfume, etc. This project involves making multiple of these tools work in a common environment to (1) evaluate them on a common dataset of logs with ground-truth models, (2) determine whether some combinations of models (e.g., the model that only accepts behavior all models accept, the model that accepts all behavior that at least one model accepts, etc.) outperform individual models.

See " Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms," (http://doi.org/10.1109/TSE.2014.2369047) as a starting point.

Automated software verification.

One of the most challenging tasks in software engineering is formal verification — proving that source code does what it is intended to do. Recent work has looked into automatically generating formal verification proofs in languages such as Coq (see, for example "Learning to Prove Theorems via Interacting with Proof Assistants" https://proceedings.mlr.press/v97/yang19a/yang19a.pdf). That paper presents the CoqGym dataset of open-source Coq projects. Expanding that dataset with more such projects would be a useful contribution (but it would require understanding and debugging Coq code, so it may be challenging if you are not familiar with this language.)

Verification-driven sound refactoring.

Suppose you have a program, and you want to refactor it to have it do the same thing but be simpler, more maintainable, etc. Create lots of mutants of the program. Identify ones that improve some useful metrics and filter out the rest. Run either an existing or a generated test suite to filter out mutants that do not behave the same way as the original program. Take the mutants that are left and (maybe automatically) use symbolic execution to verify that they are equivalent to the original program (e.g., use the Z3 theorem prover). Suggest the equivalent mutants that improve code metrics as sound refactorings that improve the code.

Fairness in software.

This is a large potential area of research. See our seminar paper that introduced the notion of testing software for bias: "Fairness Testing: Testing Software for Discrimination" (http://people.cs.umass.edu/~brun/pubs/pubs/Galhotra17fse.pdf). There has been a lot of work on improving fairness by manipulating training data, learning algorithms, and postprocessing model decisions (see "Through the Data Management Lens: Experimental Analysis and Evaluation of Fair Classification" http://doi.org/10.1145/3514221.3517841). There are many open topics here, including applying existing algorithms to common datasets for evaluation, improving fairness in learning, applying fairness testing ideas to computer vision and natural language processing, and investigating human perceptions of fairness.